// JULY/AUGUST 2015 /

Java™ magazine

By and for the Java community

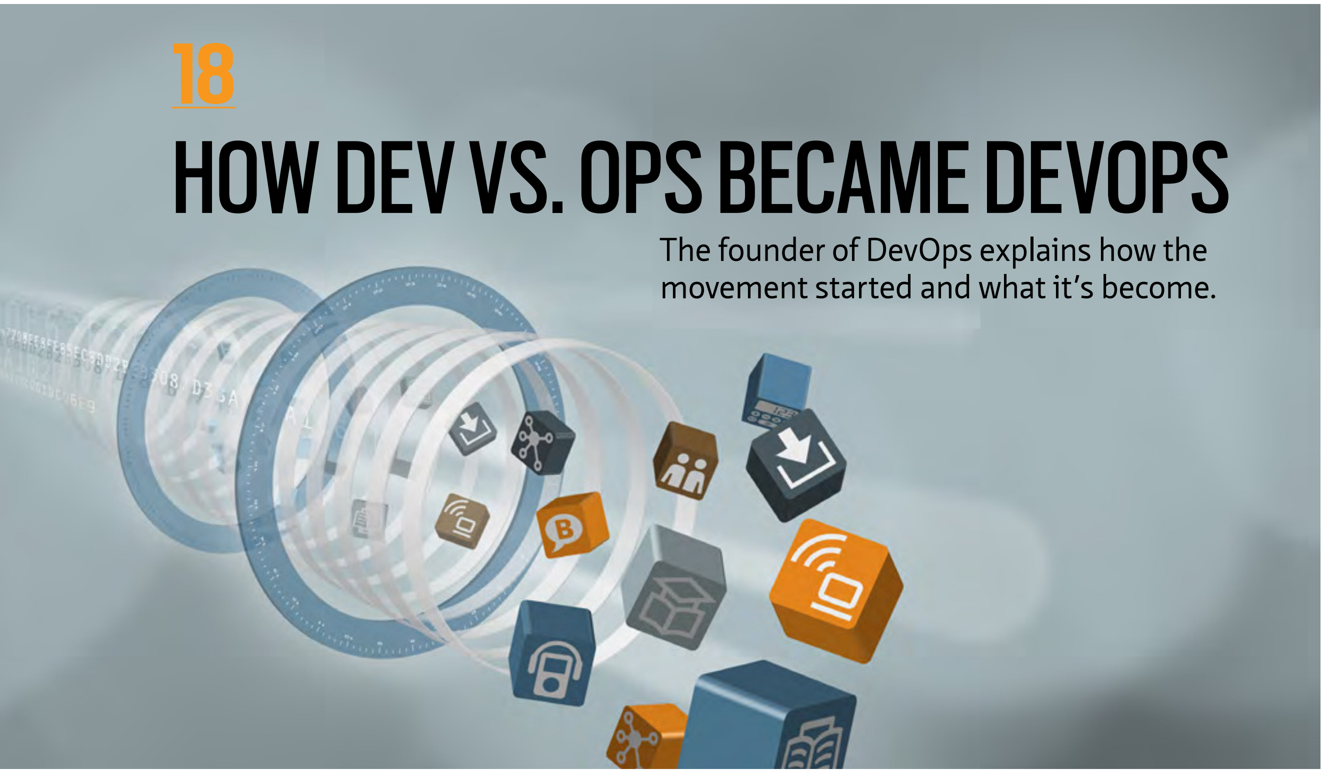
DevOps

THE PIPELINE TO BUILDING BETTER SOFTWARE FASTER

+

ORACLE.COM/JAVAMAGAZINE

ORACLE®



COVER ART BY I-HUA CHEN

EDITORIAL

Editor in Chief
Andrew Binstock

Senior Editor
Claire Breen

Copy Editor
Karen Perkins

Section Development
Michelle Kovac
Bob Larsen – Java Nation
Yolande Poirier – Java Nation

Technical Reviewers
Stephen Chin, Reza Rahman, Simon Ritter

DESIGN

Senior Creative Director
Francisco G Delgadillo

Design Director
Richard Merchán

Contributing Designers
Jaime Ferrand, Arianna Pucherelli

Senior Production Manager
Sheila Brennan

Production Designer
Kathy Cygnarowicz

PUBLISHING

Publisher
Jennifer Hamilton +1.650.506.3794

Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985

Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

President, Sprocket Media
Kyle Walkenhorst +1.323.340.8585

Western and Central US, LAD, and Canada, Sprocket Media
Tom Cometa +1.510.339.2403

Eastern US and EMEA/APAC, Sprocket Media
Mark Makinney +1.805.709.4745

Advertising Sales Assistant
Cindy Elhaj +1.626.396.9400 x 201

Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (US/Canada)

Oracle Services
+1.888.283.0591 (US)

Oracle Press Books
oraclepressbooks.com

ARTICLE SUBMISSION

If you are interested in submitting an article, please e-mail the editors.

SUBSCRIPTION INFORMATION

Subscriptions are complimentary for qualified individuals who complete the subscription form.

MAGAZINE CUSTOMER SERVICE

java@halldata.com **Phone** +1.847.763.9635

PRIVACY

Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or e-mail address not be included in this program, contact Customer Service.

Java Magazine is published bimonthly at no cost to qualified subscribers by Oracle, 500 Oracle Parkway, MS OPL-3A, Redwood City, CA 94065-1600.

Digital Publishing by GTxcel

BIO

Primitive Obsession: A Little-Discussed but Expensive Antipattern

A simple remedy brings significant benefit.

Most code smells, it seems, develop incrementally over time. Code that starts out as good goes bad by a series of subsequent maintenance passes that are not coupled with intensive refactoring. For example, smells such as long parameter lists, long functions, and God objects frequently are the product of slowly accreted functionality. Except at organizations that rigorously do code reviews, these problematic bits can live a long time in the codebase, bringing down its quality and building up technical debt.

One antipattern that survives longer than most is *primitive obsession*. This term refers to different things depending on whom you ask, but all definitions share one aspect: the use of a raw data type (a primitive) where an object would be better suited.

A classic example is using an integer to hold a US zip code (generally, a five-digit number) in a method that, say, prints addresses. If later, the program begins accepting the four-digit zip code extensions, new formatting logic will be required, because the printed form is no longer just a string of numbers. If the program then adds a zip-based barcode, additional logic will be needed. If the source code, which has been revised multiple times, remains in the original address-printing method, things get cumbersome. Imagine now, if you wanted to add postal codes for the UK, which include letters. The primitive obsession would become so obvious that the natural refactoring would be undertaken: Replace

PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

data value with object.

As a fan of small classes, I am sensitive to the opportunity to get rid of primitive obsession. Anytime a local field accretes logic specific to its operation or formatting, I move it into its own class. This step makes the parent class smaller and less complex; it also enforces the Single Responsibility Principle, which is one of the keys to effective object orientation. The new class is also easier to test and verify. And because the class now hides implementation details, changing the internal representation from an integer to a string, when the UK addresses are added, is a modification of comparatively small scope.

This data hiding is a signal benefit when the data item is a collection rather than a primitive. It can hide access to the collection's elements. For example, if I were to use a naked Java collection, rather than a wrapper object, any other method could come along and insert elements or even run `reverse()` or `swap()` without my say-so. With my wrapper, I can specify exactly what can be done to the data and so prevent all kinds of mischief.

When I return a data item, it's almost invariably immutable; and in the event an element that doesn't exist is asked for, I return an empty string, or an empty array, rather than a null. These steps mean I'm pumping cleaner data and fewer potential NPEs into the codebase. This is a lot of benefit for a simple operation.

So, if you have a data item (primitive or collection) that begins to accrete logic, move it to its own object. You'll be glad you did.

Andrew Binstock, Editor in Chief
javamag_us@oracle.com
@platypusguy

PS: Our continuing migration to a more technical, more engaging, and less promotional magazine continues. Code listings are now inline in the articles, which should greatly increase readability; we've also added a section on letters to the editor; and the books section now consists entirely of reviews, rather than promotional descriptions. Our next issue will be larger, add more innovations, and show the vision of what we want *Java Magazine* to be.

MAY/JUNE 2015

Listings Are Hard to Read

Since you asked for feedback: the listings in the articles are a nightmare. The pages (even in the PDF download) show only the Listing 1, and when I click on the other listings I'm redirected to some odd web page that either doesn't open or redirects to some general download page where I need to download all listings and search for the one I'm interested in. Takes me a minute, and by then I have forgotten what I read. The listings have to be beside the text—any other solution is complicated.

Things get worse if I'm offline: the listings in the PDF tabs don't work.

—Christian Schenk

Editor Andrew Binstock responds: Your frustration was echoed in other readers' comments and in my own experience as a longtime reader of Java Magazine. *In this issue, we've begun to address the problem of listings. To the extent possible, code is now presented inline in the articles. There will be no tabs to click on.*

One exception will be listings that are very long. These you'll need to download. With that in mind, we've renovated the download area. Listings are no longer placed in one file, but rather presented as individual code listings in plain-text files.

If you have any further suggestions or comments on these changes, please let me know. Feedback is always welcome. Information on how to contact me is available on the last page of this issue.

Confusion Over the Java char Data Type

Dear Editor,

Your review of the book *Core Java for the Impatient* by Cay S. Horstmann on page 11 of the May/June 2015 issue of *Java Magazine* has misinterpreted a Java fact. You noted a contradiction between the following two statements in the book: "Java has ... five integral types" and that in Java "four [primitive types] are integer types."

These are actually not contradictory statements. What the review has not taken into consideration is the distinction between "integral type" and "integer." While `char` is an integral type, its value is a Unicode character and not an integer. Refer to the following two excerpts from the Java 8 specification.

"The primitive types are defined to be the same on all machines and in all implementations, and are various sizes of two's-complement integers, single- and double-precision IEEE 754 standard floating-point numbers, a boolean type, and a Unicode character `char` type."

"The integral types are `byte`, `short`, `int`, and `long`, whose values are 8-bit, 16-bit, 32-bit, and 64-bit signed two's-complement integers, respectively, and `char`, whose values are 16-bit unsigned integers representing UTF-16 code units."

Notice the distinction between integral types `byte`, `short`, `int`, and `long` and the integral type `char`. Only four integral types (`byte`, `short`, `int`, and `long`) are two's-complement integers. The `char` type is an integral type but not a two's-complement integer; instead, `char` is a Unicode character.

I would suggest that an erratum be included in the next issue of *Java Magazine* about the book review being a misinterpretation of a Java fact.

—Deepak Vohra

Reviewer Andrew Binstock responds: Thanks for your note. Your citation makes my point. There is no "integer" type. It is an invention of the book's author. There are only integral types. While I don't disagree with the author that `char` *is a squirrely primitive that is distinct from its integral brethren, I don't like his creation of a new data type to make this distinction. I find this reclassification, which appears with no explanation, to be confusing, especially for readers new to Java, who are the book's target audience. I should add that in my review, I do say that this problem is a minor one and that overall I recommend this book.*

Article Series on CDI

Dear Editor,

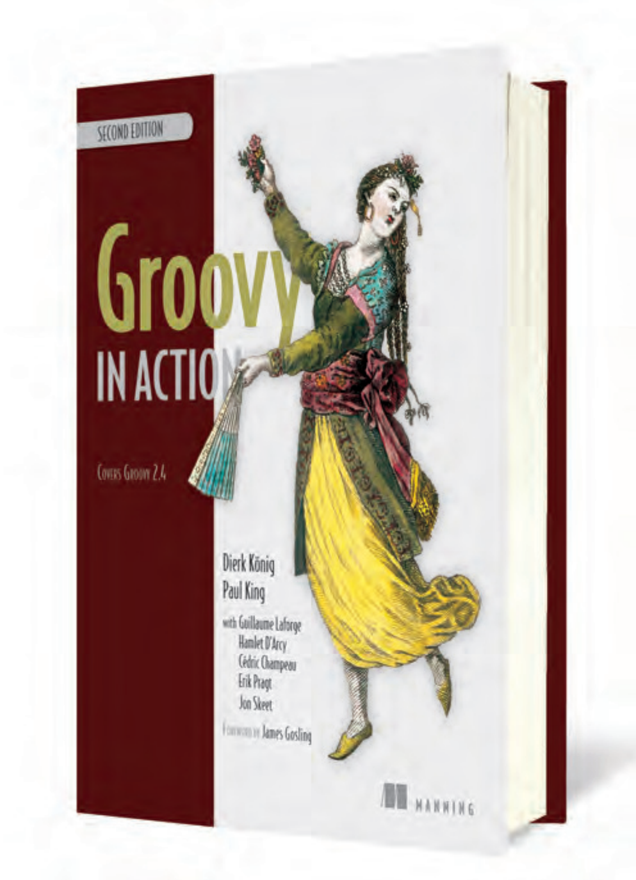
In the article "Contexts and Dependency Injection: the New Java EE Toolbox" on page 38 of the May/June issue, author Antonio Goncalves starts by showing us a `main` method that does some kind of runtime injection (using a constructor). He then shows examples of CDI injection that have nothing to do with runtime injection. But to the reader it seems as if the technique he describes could replace the runtime example. Is runtime injection something he will discuss in Part 2?

—Alexandros Trifyllis

Author Antonio Goncalves responds: I wasn't planning to talk about runtime injection, because this is an advanced topic in CDI. But now that you bring up this point, I'll try to cover it in Part 4 of the series (as Part 2 is already written [and in this issue —Ed.] *and Part 3 is nearly finished).*

Contact Us

We welcome comments, suggestions, grumbles, kudos, article proposals, and chocolate chip cookies. All but the last two might be edited for publication. If your note is private, please indicate this in your message. The last page of this issue has all our contact information.



GROOVY IN ACTION, SECOND EDITION

By Dierk König, Paul King, et al.
Manning Publications

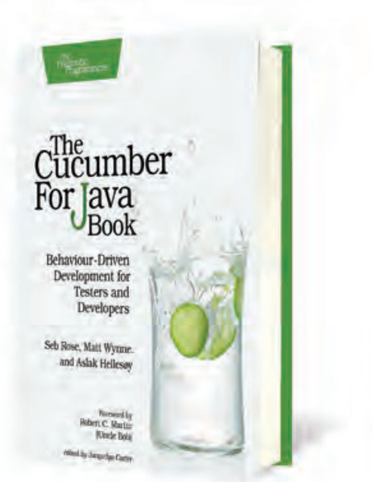
Groovy is one of the most popular and pervasive JVM languages, due to its long history as an excellent scripting idiom. It is embedded into several major tools today, including the build tool Gradle, the test framework Spock, and the web framework Grails. Part of Groovy's enduring appeal is that the language is constantly advancing with important new features added on a regular basis. Until earlier this year, that development was funded in part by Pivotal, and earlier by Pivotal's parent, VMware. Today, Groovy is an incubator project at the Apache Software Foundation.

The first edition of this book came out in 2007 and was a remarkable publication, filled with examples of extraordinary things you could do in the language with very little code. It was in all respects the bible for Groovy developers. However, due to the language's rapid development, the volume soon started to go out of date. While the examples were still valid, important new features were not covered. Since roughly 2011, publisher Manning has been trying to close this gap by having the authors work on the second edition. However, as the language kept evolving, the goalposts moved backward regularly and the wait for this second edition became an exercise in long-term patience.

This volume rewards that patience well. If you like getting deep into programming languages and working on challenging problems, this book provides ample rewards. Naturally, it has a full explanation of the language. But it quickly moves on to explaining how to do hard things simply. For example, one of the brilliant aspects of Groovy is its ability to provide access to abstract syntax trees it creates when compiling source code. If you want to inject your own transformations, you can do so with comparative ease. This book spends more than 60 pages illustrating how to do this. Like doing metaprogramming (that is, intercepting and modifying how Groovy objects are called)? Thirty pages of explanations are presented. And so on.

There is also a very interesting discussion of one of Groovy's best-known features: its ability to quickly script unit, integration, and acceptance tests. JUnit is built in to Groovy. However, the Groovy testing library includes many very handy assertions, full mocking capabilities, and dynamic class creation. It's the complete testing bundle.

The authors of this book are all major contributors to the Groovy ecosystem and have done an excellent job explaining the language's features and how best to leverage them in advanced, often complex scenarios. By quality and size (850 pages), this book can certainly reclaim the title of being the one must-have book on Groovy programming. Heartily recommended.
—*Andrew Binstock*



THE CUCUMBER FOR JAVA BOOK

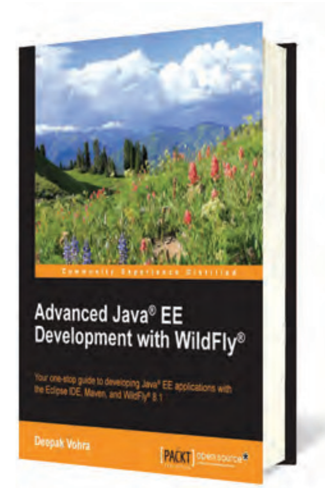
By Seb Rose, Matt Wynne, and Aslak Hellesoy
The Pragmatic Bookshelf

Cucumber is a tool, originally written in Ruby, with which you can capture user requirements in text and make them the basis for acceptance tests. This book covers the popular tool's Java instantiation, Cucumber for Java, and presents this approach to testing code, frequently termed behavior-driven development (BDD). The driving ideas behind BDD are twofold. The first is to enable the conversation between users and developers by capturing the former's requirements in "given x, if y happens, then z should be the result" scenarios. A domain-specific language (DSL) is used to construct these scenarios, which are then converted into tests that the BDD tool can then run automatically. Developers use the DSL to specify which functions/methods to call to fulfill the script and how to test the result. Scenarios for which no code yet exists are counted but not reported as errors when the tool, in this case Cucumber, runs the tests and reports the results. Errors, of course, are reported as such.

The second goal of BDD is to enable developers to make sure they're always testing against what the user has asked for. If you've used test-driven development, then the idea of having tests define coding success in process will be familiar. Here, however, the tests are at a much higher level—user-acceptance tests—but they make an excellent complement to unit tests.

Gherkin is the DSL for capturing tests, and Cucumber is the framework that drives them. This book explains Cucumber and Gherkin, and introduces BDD both as a mindset and as a pure testing technique. It is a clear and cogent exposition of the features that details in depth how to write tests and tie them to Java code. Subsequent chapters dive into specific challenges, such as testing scenarios involving databases or message queues and, of course, web apps and APIs. Unfortunately, mobile apps are not covered.

The well-written, approachable text is supported with sensible examples and clear code. If you want to try out BDD, this book is an excellent place to start. —*A.B.*



ADVANCED JAVA EE DEVELOPMENT WITH WILDFLY

By Deepak Vohra
Packt Publishing

Among top-tier technical publishers, you can safely rely on a base level of quality. With other publishers, such as Packt, this is not the case. You must check out prospective purchases carefully.

This advice applies to the present book, which delivers in part on the title's promise, but does so within very narrow limits. I'd be hard-pressed to view most of the material as "advanced." It is mostly bread-and-butter Java EE programming. Challenging issues, such as testing, are not covered.

To use this book, you will need WildFly 8.1 (Red Hat's renamed JBoss Java EE server), Eclipse (and only Eclipse), Maven (and only Maven), and MySQL. Given that lineup, the author takes you through a short set of recipes: one each on Hibernate, JavaServer Faces, Ajax, GWT, a JAX-WS web service, and finally Spring 4.1. The text swells to 400 pages due to heavy use of screenshots of Eclipse dialog boxes. The images are accompanied by do-this, do-that instructions, rather than tutorials. Inexplicably, the book also prints entire pom.xml files, including one that stretches for more than seven pages.

While the author shows you how to use the technology, he occasionally veers away from best practices. The chapter on Hibernate has a section on creating an XML mapping file. Nowhere does he mention that mapping files are outdated resources, or that annotations have been the preferred solution for years. In fact, this section never mentions annotations.

Finally, the code has been incorrectly typeset in various places so that some code won't compile as printed.

This book can be recommended only to readers who are searching for very specific WildFly-based recipes, who can work strictly in Eclipse, and who are willing to overlook its errors. —*A.B.*

DevOps:
It's Not Just For The Unicorns

There's a stigma that exists where we constantly see suggestions that DevOps is just for the unicorns. This couldn't be more false. Afterall, those unicorns were once horses themselves.

Ready to start your journey to becoming a successful DevOps IT organization?

Download your copy of "Making The Shift: From Continuous Integration to Continuous Delivery" and start automating your development and delivery pipeline.

CloudBees®
The Enterprise Jenkins Company

//events /


Santa Clara, California

JVM Language Summit *AUGUST 10–12*
SANTA CLARA, CALIFORNIA
The JVM Language Summit is an open technical collaboration among language designers, compiler writers, tool builders, runtime engineers, and VM architects.

JCertif 2015
AUGUST 14–16; KINSHASA, REPUBLIC OF THE CONGO
SEPTEMBER 7–13; BRAZZAVILLE, CONGO
JCertif International has trained more than 5,000 new Java developers in Africa since it began, and its conferences are expanding to more countries. The events usually include two- to five-day Java workshops for 20 to 50 developers. This year's topics include Java; Android; and wearable, object-connected, and drone technologies.

JavaZone
SEPTEMBER 9–10
OSLO, NORWAY
JavaZone, created by the Norwegian Java User Group (javaBin), consists of almost 200 speakers, 2,500 attendees, and seven parallel sessions. An additional day of workshops beforehand is included in the cost of the ticket.

JDayLviv
SEPTEMBER 19
LVIV, UKRAINE
Enjoy and learn in a full day of world-class talks. Topics include JVM languages, cloud and infrastructures, big data and high load, architecture, and front end.

code.talks 2015
SEPTEMBER 29–30
HAMBURG, GERMANY
More than 1,500 participants take part in 110 presentations across 13 tracks, including Java, PHP, JavaScript, DevOps, big data, e-commerce, mobile, and more.

Silicon Valley Code Camp
OCTOBER 3–4
SILICON VALLEY AREA, CALIFORNIA
Last year, 4,500 people attended this free community event where developers learn from fellow developers. In addition to technical topics, speakers present on software branding and legal issues.

JAX London
OCTOBER 12–14
LONDON, ENGLAND
JAX London brings Java, JVM, and enterprise professionals together for a technology- and methodology-packed event. Participants get full access to Big Data Con London, which features modern datastores, big data architectures based on Hadoop, and advanced data processing techniques.

JDD
OCTOBER 12–13
KRAKOW, POLAND
JDD is a two-day conference for all Java enthusiasts, who can participate in more than 30 lectures, workshops, interactive trainings, and networking opportunities. JDD attracts speakers from all over

PHOTOGRAPH BY RICHARD CUMMINS/CORBIS

the world. It also offers lectures in English.

GeeCON

OCTOBER 23–24
PRAGUE, CZECH REPUBLIC
Join more than 2,000 participants at GeeCON, which is focused on JVM-based technologies, with special attention to dynamic languages such as Groovy and Ruby. GeeCON participants share experiences about development methodologies and craftsmanship, enterprise architectures, design patterns, distributed computing, and more.

JavaOne 2015

OCTOBER 25–29
SAN FRANCISCO, CALIFORNIA
Join the single largest gathering of Java developers. From sessions, workshops, labs, and exhibits to keynotes and Birds-of-a-Feather sessions, learn about the latest language changes to improve coding efficiency. You'll also learn how to build modern enterprise and server-based applications, create rich and immersive client-side solutions, build next-generation apps targeting smart devices, and compose sophisticated Java web services and cloud solutions.

W-JAX 15

NOVEMBER 2–6
MUNICH, GERMANY
The W-JAX conference covers current and future-oriented technologies from Java, Scala, Android, and web technologies to agile development models and DevOps.

Devoxx Belgium

NOVEMBER 9–13
ANTWERP, BELGIUM
By developers for developers, this event has 200 speakers and 3,500 attendees from 40 countries. Tracks this year include Java SE, JVM languages, and server-side Java, as well as cloud and big data, mobile, and architecture and security, among others.

Devoxx Morocco

NOVEMBER 16–18
CASABLANCA, MOROCCO
Formerly the JMaghreb conference, this event is a university day of training, workshops, and labs followed by conference days of sessions on software development, web, mobile, gaming, security, methodology, Internet of Things (IoT), and cloud. The Decision Makers evening includes discussion of issues related to the IT industry in Morocco.

Groovy and Grails eXchange 2015

DECEMBER 14–15
LONDON, ENGLAND
Stay ahead of the curve and hear the 2016 roadmap for Groovy and Grails from core committers and Groovy authorities Guillaume Laforge and Graeme Rocher. Engage with other leading experts and fellow enthusiasts and learn the latest innovations and practices.

Have an upcoming conference you'd like to add to our listing? Send us a link and a description of your event at least four months in advance at javamag_us@oracle.com. We'll include as many as space permits.

PHOTOGRAPH BY JANETTE ASCHE/GETTY IMAGES

Quiz Yourself

You never create technical debt. It's just that other developers don't really know the details of how Java works. We completely understand.

The questions in this quiz section are taken from the certification test 1Z0-808: Oracle Certified Associate, Java SE 8 Programmer, Oracle Certified Java Programmer. (The purpose of this certification is to enable beginners to demonstrate their knowledge of fundamental Java concepts. The certification is designed for beginners in programming and/or those with a nonprogramming background who have basic mathematical, logical, and analytical problem-solving skills and who want to begin to learn the Java programming language; and for novice programmers and those programmers who prefer to start learning the Java programming language at an introductory level. After successfully completing this certification, candidates can confidently utilize their knowledge on Java datatypes, operators, statements, arrays, lists, and exception-handling techniques.)

Naturally, these questions have small embedded traps that spring open if you rush through without considering exactly what the code is doing. Ready? (Answers appear in the "Solutions" section after the questions.)

Question 1. Given this code fragment:

```
List<String> vitamins = new ArrayList<>();
vitamins.add("A");
vitamins.add("B12");
vitamins.add("C");
vitamins.set(1,"B");
vitamins.add(1,"D");
System.out.println(vitamins);
```

What is the result?

a. [A, D, B, C]
b. [A, D, C]
c. [D, B, C]
d. [D, B12, C]

Question 2. Given this code fragment:

```
class Engine { }
public class App {
    public static void main(String[] args) {
        Engine e = new Engine();
        Engine e1 = e;
        e = null;
    }
}
```

Which statement is true about this code?

a. It creates an object and the object is eligible for garbage collection.
b. It creates an object and the object is *not* eligible for garbage collection.
c. It creates two objects: `e` and `e1`. The `e` object is eligible for garbage collection.
d. It creates two objects and both the objects are *not* eligible for garbage collection.

Question 3. Given this code fragment from Cart.java:

```
package shop;
// line n1
{
  Cart() {
    System.out.println("Cart is created.");
```

```
   }
}
```

And this content from Shop.java:

```
package shop;
public class OnlineCart extends Cart{
   public static void main(String[] args) {
     Cart c = new OnlineCart();
   }
}
```

Which code fragment can be inserted at `line n1` to enable the `Shop` class to compile?

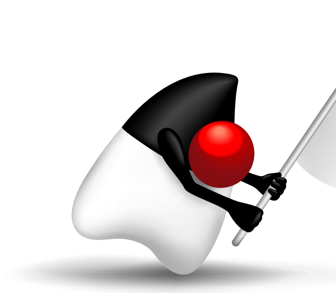
a. `final class Cart`
b. `public class Cart`
c. `private class Cart`
d. `protected class Cart`

Question 4. Given this code fragment:

```
double wage = 2.0;              // line n1
int weekDays = 5;
long monDays = weekDays * 4;       // line n2
long yearDays = monDays * 12L;     // line n3
long totalWage = yearDays * wage; // line n4
```

Which modification enables the code to compile?

a. Replace line nl with `double wage = 2;`
b. Replace line n2 with `long monDays = weekDays * 4L;`
c. Replace line n3 with `long yearDays = monDays * 12;`
d. Replace line n4 with `long totalWage = yearDays * (long) wage;`



Solutions

Question 1. The correct answer is A. Explanation:
`java.util.List` is an index-based collection. The `set()` method replaces the element at the specified position in the list with the specified element. The `add()` method inserts the specified element at the specified position in the list.

Question 2. The correct answer is B. Explanation:
At line 4, the code creates an object that is accessible by using the `e` reference variable.

At line 5, `e1` is another reference variable for the same object and there are two reference variables for the `Engine` object. Hence, the object is accessible by using both the `e` and `e1` variables.

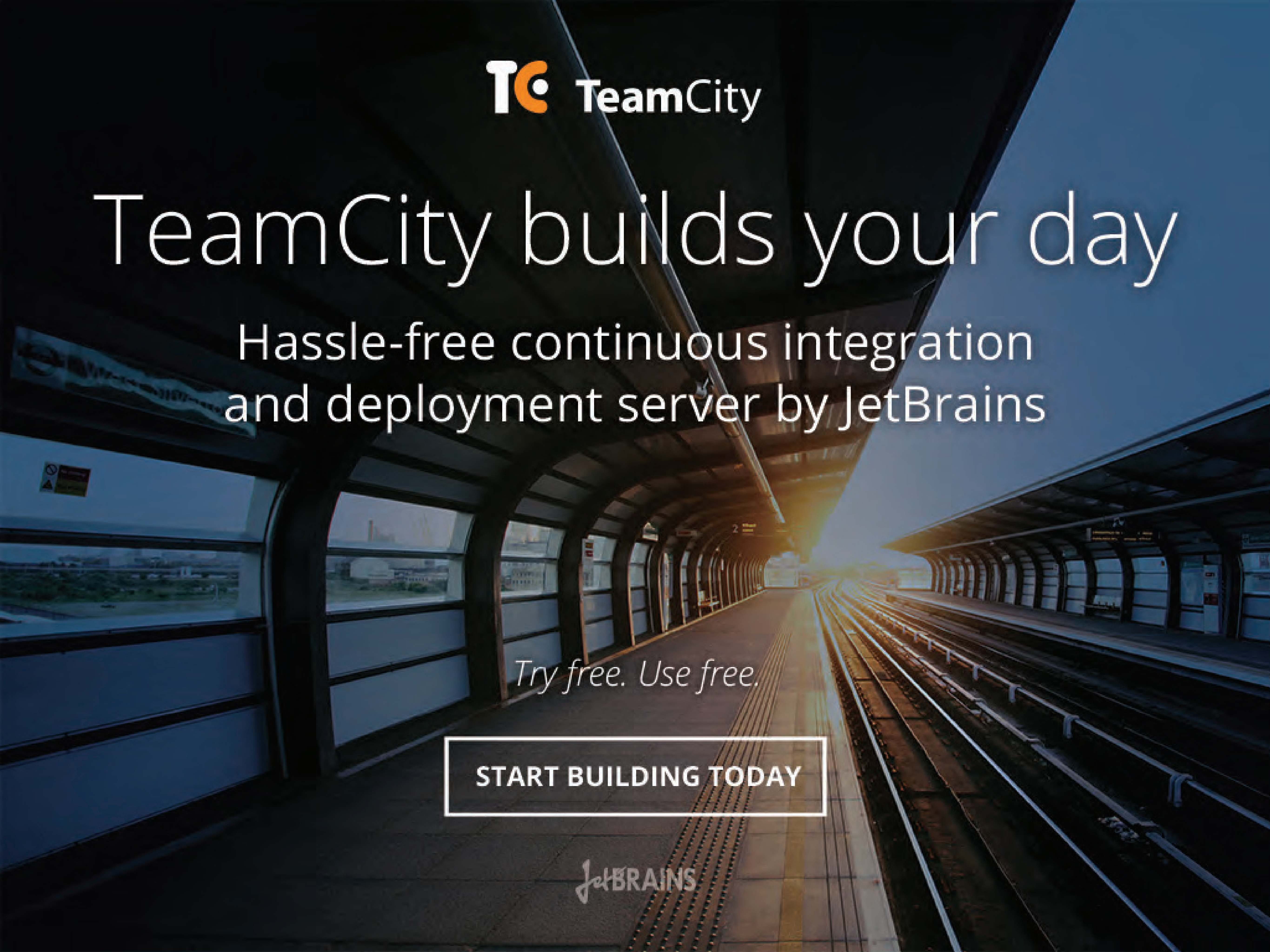
At line 6, the assignment of `null` to the `e` reference variable makes the object inaccessible through `e`.

Option A is incorrect because the object is accessible through the `e1` reference variable. The inaccessible object is eligible for garbage collection. Options C and D are incorrect because only one object is created.

Question 3. The correct answer is B. Explanation:
Option A is incorrect because a `final` class cannot be inherited. Option B is correct because the `public` classes are accessible and inheritable. Options C and D are incorrect because the top-level class, `Cart`, cannot be declared with a `private` or `protected` access modifier.

Question 4. The correct answer is D. Explanation:
Option D is correct. Line n4 fails to compile due to loss of precision. To fix the error, the `wage` variable must be cast to the `long` type. Option A is incorrect. It is a valid assignment, hence the modification suggested in Option A does not meet the requirement. Options B and C are incorrect. They are valid assignments. The suffix `L` converts to `long` type.

[Our gratitude to Sushma Jagannath of Oracle University for her kind help in providing these test extracts. —Ed.]


TeamCity
TeamCity builds your day
Hassle-free continuous integration
and deployment server by JetBrains
Try free. Use free.
START BUILDING TODAY
JetBRAINS

DevOps: BUILDING THE PIPELINES

When conceived by its creator, Patrick Debois, DevOps was intended to bring developers and operations (that is, system administrators) into closer cooperation. As he states in our interview (page 18), DevOps has given developers more insight into how their code runs and helped admins become less risk-averse. As Netflix shows (page 22), substantial investments in a comprehensive DevOps pipeline can provide those benefits and other unexpected insights. However, when we look at details of pipeline management by using Docker (page 28) or Chef (page 35), it's clear that for small to midsize organizations, DevOps can represent a lot of new work, particularly for developers. A cloud orientation will help, as the needed pipelines are easier to set up and manage for devs and admins.

This cloud dimension pervades not only these articles but DevOps at large. The transition from continuous delivery, as a bleeding-edge extension to continuous integration, to DevOps maps almost directly to the transition from one-off virtualization to the large-scale cloud as a platform. In this sense, DevOps is completely new. —*Andrew Binstock*

ART BY I-HUA CHEN

Patrick Debois (middle), who coined the term *DevOps*, currently works in a consultancy in Ghent, Belgium.

DevOps

HOW DEV VS. OPS BECAME DEVOPS

The founder of DevOps explains how the movement started and where it's headed.

BY STEPHEN CHIN

PHOTOGRAPHS BY TON HENDRIKS

Widely known as the father of the DevOps movement for his work in bootstrapping the influential DevOpsDays conferences, Patrick Debois recounts the origins of the term, the concepts, and the subsequent growth of what is now standard practice at many large developer IT organizations.

Java Magazine: Tell us a little bit about how you came to found DevOps.

Debois: Six or seven years ago, I was working on a project that involved agile development. Although I was a database admin, I really liked the flexibility of the agile mentality and tried to translate everything that agile development was doing into my systems work—such as faster feedback, testing at the infrastructure level, and so forth. We called what we were doing at the time "agile system administration."

Java Magazine: Not nearly as cool sounding as "DevOps."
Debois: It was also narrowly focused, because it only included database and system administration. Then I went to the 2008 Agile Toronto conference and met Andrew Shafer [@littleidea on Twitter], a software developer who was already talking about agile infrastructure. We started the Agile System Administrator Google Group. Later, I saw a talk by Jean-Paul Sergent about Dev and Ops working together. I didn't really think about the term DevOps at that time, but I mentioned on Twitter that it would be cool to have a conference in Europe exploring some of these ideas. "Agile System Administrators" isn't a very good name for a conference, so Sergent suggested we call it "DevOpsDays." And that is how the DevOps movement started.

"If you're not willing to collaborate, it doesn't really matter what tool you use or don't use."

Java Magazine: So the conference name came before the term *DevOps*?

Debois: Yes. We used Twitter to connect with about 60 people in Belgium. Two things were memorable about that conference. One was the energy of the conference people. Secondly, it was very international. Several attendees flew in from Australia and the US. Others came from France, Germany, and the UK. We started with just a couple of tweets about something that didn't exist, and we had 60 people from all over the world come to that conference. After the conference, we continued our discussions using the hashtag #DevOps.

Java Magazine: Despite your initial Ops perspective, a lot of DevOps traction has come from developer interest. What attracts developers to DevOps?
Debois: I came from the admin side, but I always had an integration role. In the early days when Java didn't have servlets, I created my own servlet loaders because I needed to have functioning websites. I had an affinity for developers because it always struck me that when you're in the developer role, you often hate the operational constraints. In discussions about infrastructure, admins would throw comments at you, such as, "No, it's not the correct directory. You need to change the ports." It's really annoying that you can't handle that yourself.

Java Magazine: Hence the need for Dev and Ops integration.
Debois: Before the DevOps movement, once your app was running, you didn't have any insight into what else was going on. You couldn't see the logins, you didn't have access to the file system if something failed, and you couldn't do debugging. You couldn't see how see how things ran in production in order to avoid making the same mistakes. Those were the main drivers that led me to believe that all this needs to be more flexible. Why can't developers and operations collaborate instead of hitting a brick wall? That was the incentive most developers I worked with had, and I think most people have.

If you're really passionate about what you're building and you want to improve things, you do want to go that extra mile, and you do want to have that information, and you do feel responsible about how your application is doing in production.

Java Magazine: Much of your focus has been on how people interact, as well as streamlining how developers and operations work together. In the past five years, there have been a lot of advances in tooling associated with DevOps. Are these tools changing the way that people do development, or have they become too much of a crutch?
Debois: There is always pushback when you try to change culture, and a tool by itself can't change culture. What's good

"In general, a developer assumes the world doesn't change outside his app, and an admin assumes the opposite."

about a new tool or a new programming language is that it makes you think differently about your problem.

A good example of this is virtualization. In the beginning, people were using it in much the same way as when they didn't have virtualization. But once you start thinking deeply about virtualization as disposable and rebuildable environments, your concept of virtualization changes. One of the things that good tools can do is give you faster feedback on things that you or someone else is doing—feedback that can be captured via a simple chat system or as part of your testing system.

Tools that improve feedback also boost collaboration. You might want to collaborate, but if it takes a week to get a report about a problem, you feel less incentive to work on it. It's what's in your immediate attention span that keeps you going. The risk, of course, is if you're doing continuous integration or continuous delivery, you take it to a point where you're not getting feedback from your customers.

Java Magazine: That's pushing it to the extreme if you're not getting feedback from your customers.
Debois: That is an agile concept that was missing in practice. Operations was still finding anomalies and saying somebody else in the pipeline was responsible, which shouldn't be the case. Tools that get information to the developers, such as dashboards and monitoring systems, all help provide feedback and shorten the cycle. They support your process tremendously if you are willing to collaborate. Of course, if you're not willing to collaborate, it doesn't really matter what tool you use or don't use.

Java Magazine: How has DevOps changed the views of admin teams?
Debois: DevOps has changed the culture in the admin world. Admins used to be risk-averse. Their mantra used to be "we don't do this." Now, admins are saying "let's do it more often if it's hard," "let's see what happens when we try to automate," or "let's feel the pain and then discuss things because now we're an equal stakeholder in the discussion." For developers, who have always been at the center of feature discussions and product decisions, that culture was already there if they were doing agile. But within an operations group or a systems group, this was not a common mentality.

Java Magazine: In the early days of extreme programming, we were pushing the development envelope and people were averse to making changes because they worried that more-frequent releases would introduce more bugs. Now, it's completely shifted in the other direction to where most organizations are embracing agile development. It sounds as though DevOps is having the same kind of cultural effect on operations groups by making drastic changes in how applications are deployed.
Debois: This change comes partly from the fact that, in modern infrastructures, we're dealing less with hardware and more with an API or programmable infrastructure. So admins have evolved to thinking more programmatically about infrastructure, which is a natural fit with the current development process. Feedback cycles are often still too slow: testing on Ubuntu and creating a VM [virtual machine] is much slower than doing something locally, although technologies such as Docker are improving the feedback cycle.

But it's complex. In general, a developer assumes the world doesn't change outside his app, and an admin assumes the opposite: he owns the app and has to keep it running. An admin doesn't have the skills to change the app, but he knows the world is changing and so he deals with that part of the world that involves protecting the app or making sure something is patched.

Java Magazine: Are there other disciplines where you can apply the best practices and experience you've gained from DevOps?
Debois: Interesting question. I deliberately made the decision of not working for another big technology company. I didn't want to be in the back room; I wanted to be in on product discussions. I realized that I could use all the technology knowl-

"Mobile testing is hard, but it's going to get much harder if the Internet of Things takes off."

edge I've garnered, and all the friends in the DevOps space, to make a difference in another domain. I am currently in the broadcast video and live show space, which has interesting restrictions. One is the fact that you don't have any users two minutes before the show, and then when the show is aired, you have tens or hundreds of thousands of users. Then after the show they're gone, so you don't have a test audience. You only have that one hour, so how do you do your testing?

It's like Formula One auto racing, where you go to a pit stop when you need a fix. I'm monitoring this in real time because, if it takes five minutes, your audience will miss five minutes of the show, which might be an issue.

Java Magazine: By that time, you've probably lost your audience.

Debois: Yes. There's technology complexity, but the real challenge is the content silo. The people who make the show's content are allowed to dream. I don't give them arguments on what's possible or not. I try to solve the issues. It's an interesting dialogue between content and producers, not unlike that between Dev and Ops. I see feedback during the show because we're on chat and I can monitor our business counters for how many hits or how many posts, so it is just another way of thinking about fast feedback and collaboration during a show.

Java Magazine: Could the feedback you're getting from video production and live audiences be brought into future DevOps discussions?

Debois: I used to do web development, where we had to scale up our back-end servers to monitor failures our end users were experiencing, which depended on all the different types of mobile phones they carried. The kind of testing I'm doing now is easy compared to that.

Mobile testing is hard, but testing is going to get much harder if the Internet of Things ever takes off. The sheer variety of devices, and the need to test all these networked things to keep them updated and humming, will make for a fascinating set of challenges and opportunities. </article>

Stephen Chin is an evangelist for Java technology at Oracle and a frequent contributor to *Java Magazine*.

LEARN MORE

- *Continuous Delivery*, the book that led the way to DevOps
- A discussion of what DevOps is and is not

CONTINUOUS DELIVERY OF MICROSERVICES AT NETFLIX

By innovating its development pipeline to accommodate microservices, Netflix automates updates and rapidly tests new ideas. **BY SANGEETA NARAYANAN**

In today's marketplace, companies need to innovate continuously in order to stay relevant and sustain growth. Consequently, velocity becomes a primary requirement for software engineering organizations. To move fast, systems need to be architected with agility in mind. For such systems, the actual process of software delivery can be fully automated, as we have done at Netflix by using continuous delivery to roll out new versions of our microservices.

Microservices are an architectural approach in which a single application is built from a suite of small, collaborating services. Each service is responsible for a subset of the application functionality and can be operated independently of the others.

Continuous delivery (CD) is a software development practice that makes it cheap, quick, and easy to roll out new versions of an application with confidence. The idea is to develop a software delivery process that allows every commit to automatically be deployed to production, with visibility into the entire process.

A combination of microservices and CD enables companies such as Netflix to rapidly test new ideas and continuously improve the customer experience.

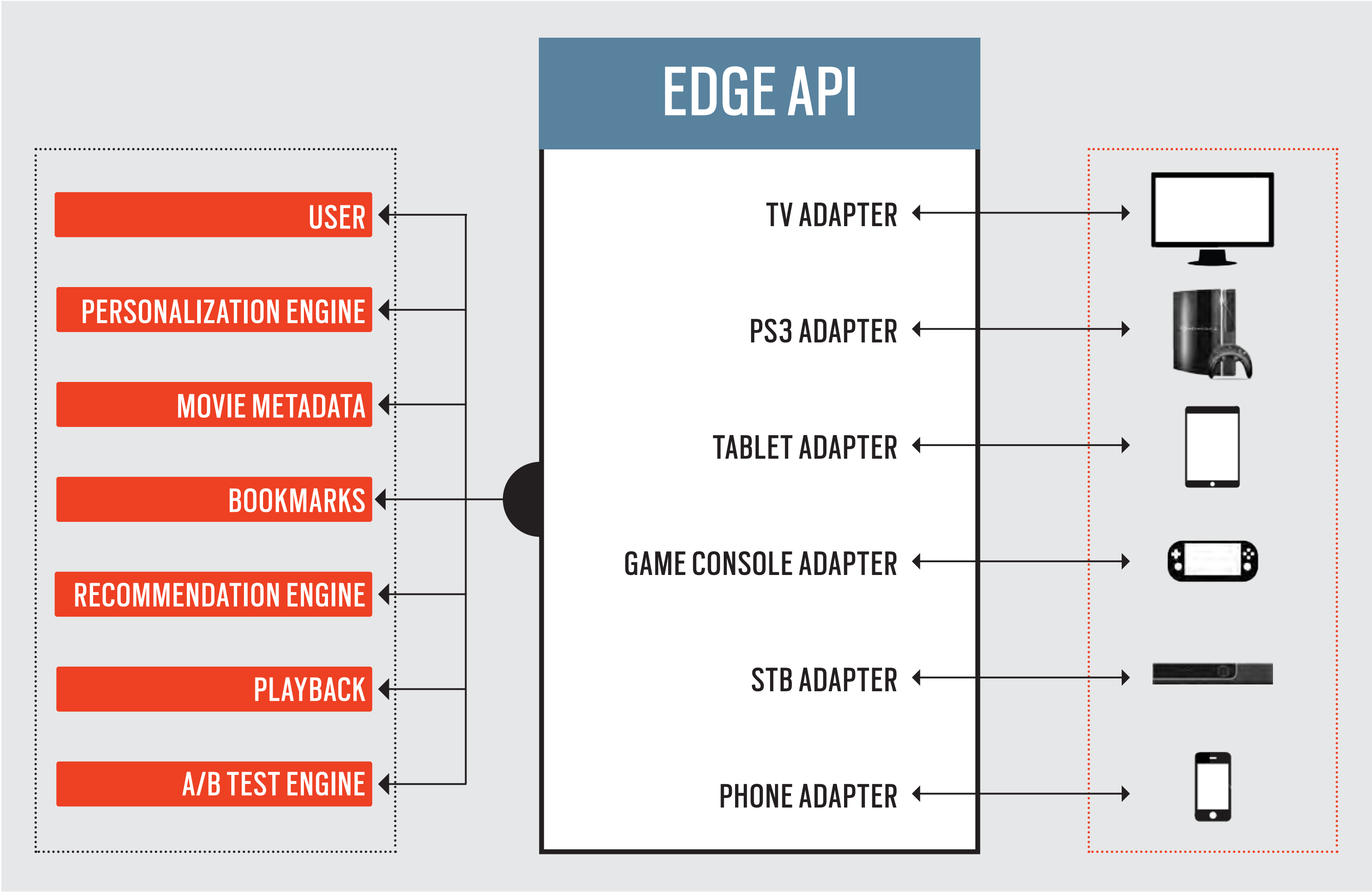


Figure 1. The Netflix Edge API Service

THE NETFLIX EXAMPLE

The Netflix product consists of a set of microservices running on the Amazon Web Services (AWS) cloud, each focused on a specific business function. Subscribers worldwide watch Netflix on more than 1,000 device types. The Netflix Edge API service powers the user experience by connecting subscribers' devices to the Netflix streaming ecosystem in AWS.

The API, like a majority of Netflix microservices, is a Java application executing within an Apache Tomcat container. Communication between devices and the API occurs over HTTP, as does the interaction between the API and the other services.

Figure 1 provides an overview of the Netflix streaming service. Each box in this image represents a component that is developed and operated independently. This decoupling is a key enabler of agility. The sections that follow touch upon a few CD concepts using the API process as an example.

THE NETFLIX EDGE API DELIVERY PIPELINE

Delivery pipelines are an important concept in the practice of CD. A pipeline consists of stages through which code passes, eventually resulting in a deployable artifact. Each stage provides feedback about code quality and deployment readiness. A failure in any stage halts the pipeline, and no code can move through the system until the failure is corrected.

The API delivery pipeline is illustrated in **Figure 2**.

Development and testing. Every commit starts out as a release candidate, so it is important to select a development workflow optimized for CD. The API workflow is modeled after GitHub Flow, with one variation: pull-request processing has been automated to run a test suite in addition to allowing for optional manual code reviews. This ensures that the code won't break the master build.

Automated tests are the foundation of a delivery pipeline. For us, unit tests and integration tests—coupled with automated canary analysis (discussed later)—form the core of the test strategy for the Netflix Edge API.

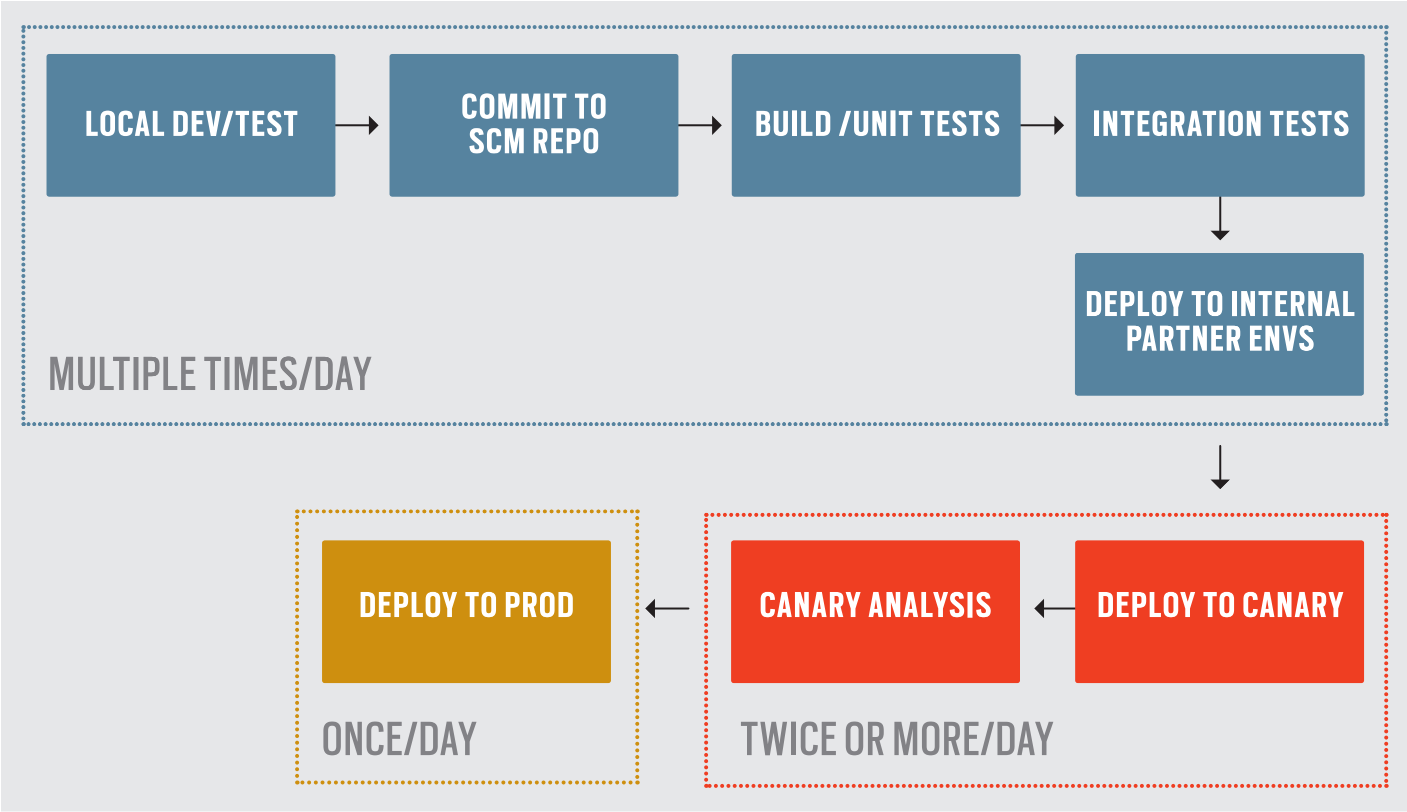


Figure 2. API Delivery Pipeline

Dependency management. Dependency management is an important consideration when building Java applications. The ideal dependency management strategy balances the need for generating reproducible builds while allowing for rapid dependency updates.

The Netflix Edge API has a complex dependency graph based on hundreds of libraries, most of which are being updated continuously. Gradle is used for build automation and dependency management, along with a few custom plugins that have been open sourced under Nebula. The gradle-dependency-lock plugin resolves dependencies based on dynamic versions and allows locking of dependency versions based on the resulting graph. Here is a snippet from a `build.gradle` file that is configured to always pull the latest released version of its dependencies:

```
apply plugin: "nebula-grails"
apply plugin: "nebula-ospackage-tomcat"
apply plugin: 'gradle-dependency-lock'
. . .
dependencies {

    provided
       "javax.websocket:
           javax.websocket-api:latest.release"
    runtime
       "javax.servlet:jstl:latest.release"

    compile "com.fasterxml.jackson.core:
        jackson-annotations:latest.release"

    test "org.grails.plugins:
        code-coverage:latest.release"
   }
```

Running `$./gradlew generateLock` generates `dependencies.lock`, which contains concrete versions for the dependencies.

```
{
"javax.websocket:javax.websocket-api": {
    "locked": "1.0",
    "transitive": [ "netflix:chronos-client",
       "org.eclipse.jetty.websocket:
           javax-websocket-client-impl",
       "org.eclipse.jetty.websocket:
           javax-websocket-server-impl" ] },
"javax.servlet:jstl":
    { "locked": "1.2", "requested": "latest.release" },
"com.fasterxml.jackson.core:jackson-annotations":
     { "locked": "2.5.3",
       "requested": "latest.release",
       "transitive": [
          "com.fasterxml.jackson.core:jackson-databind",
          "com.netflix.api.insights:insights-common",
          "com.netflix.archaius:archaius-core",
          "com.netflix.suro:suro-client",
          "com.netflix.suro:suro-core",
          "com.netflix.suro:suro-kafka-producer" ] },
"org.grails.plugins:code-coverage":
     { "locked": "2.0.3-3",
       "requested": "latest.release" }
}
```

The following command causes tests to be executed to validate the generated set of dependencies:

```
$./gradlew -PdependencyLock.useGeneratedLock=true test
```

If the tests pass, the `.lock` file is saved and committed to the repository.

```
$./gradlew saveLock commitLock
```

Using a locked set of dependencies for the API insulates developers and the production pipeline from churn in library versions. A separate instance of the pipeline updates `dependencies.lock` on a regular schedule by running the previous steps asynchronously. The plugin includes several other powerful features, such as selective update of libraries, overriding library versions to force to a particular version, and different ways to manage transitive dependencies.

Automated canary analysis. *Canary testing* is the process of rolling out a new version of software to a small set of end users with the goal of having them serve as an early warning system, much as miners in the old days used canaries to detect the presence of toxic gases in mines. Netflix has built a sophisticated canary analysis system that generates a "readiness signal" based on the comparison of a set of metrics across two versions of an application. This system is integrated into the API delivery pipeline to automatically trigger a production deployment if the readiness score is above a certain threshold.

The key to a successful canary process is identifying the set of metrics that will generate an accurate signal. API developers instrument the code to publish metrics that will be useful in understanding system behavior. Any metric can be included in the canary analysis, but developers control the overall signal by identifying the most-relevant metrics, weighting them appropriately, and determining the acceptable level of deviations for each.

Figure 3 illustrates a successful canary that was deployed to production. The "system" category had some deviations, but they were not weighted highly enough to bring the overall score below the readiness threshold.

Figure 4 provides details of the metrics that contributed to the score for that category.

Multiregion deployment and rollback. A build that has passed the canary stage is ready for deployment. An important concept here is that of *immutable deployments*, where the infrastructure is based on a set of components that are replaced rather than updated for each deployment. An Amazon Machine Image (AMI) can be leveraged to accomplish immutable deployments to AWS.

AMI-based deployments enable the use of the Red/Black deployment strategy for the API. A new server farm is launched with the latest AMI alongside the old farm. Once the new farm (Red) is scaled up appropriately, the old one (Black) is disabled. This process is repeated for all the AWS regions in which Netflix operates. **Figure 5** illustrates how this works for one region.

The old server farm is retained for a few hours to facilitate a quick rollback in the event of a major issue with the new code. Rollback is also automated and takes into account scaling up the old cluster as required. All changes to produc-

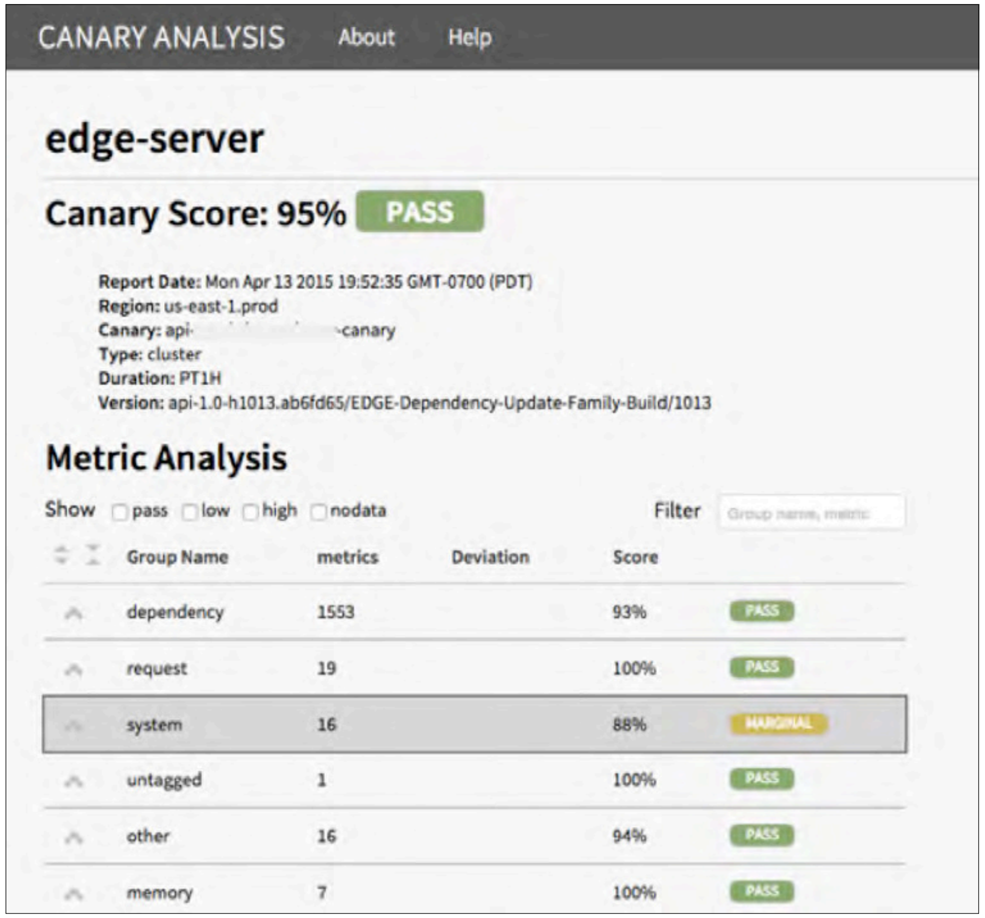


Figure 3. Canary Analysis Report Summary

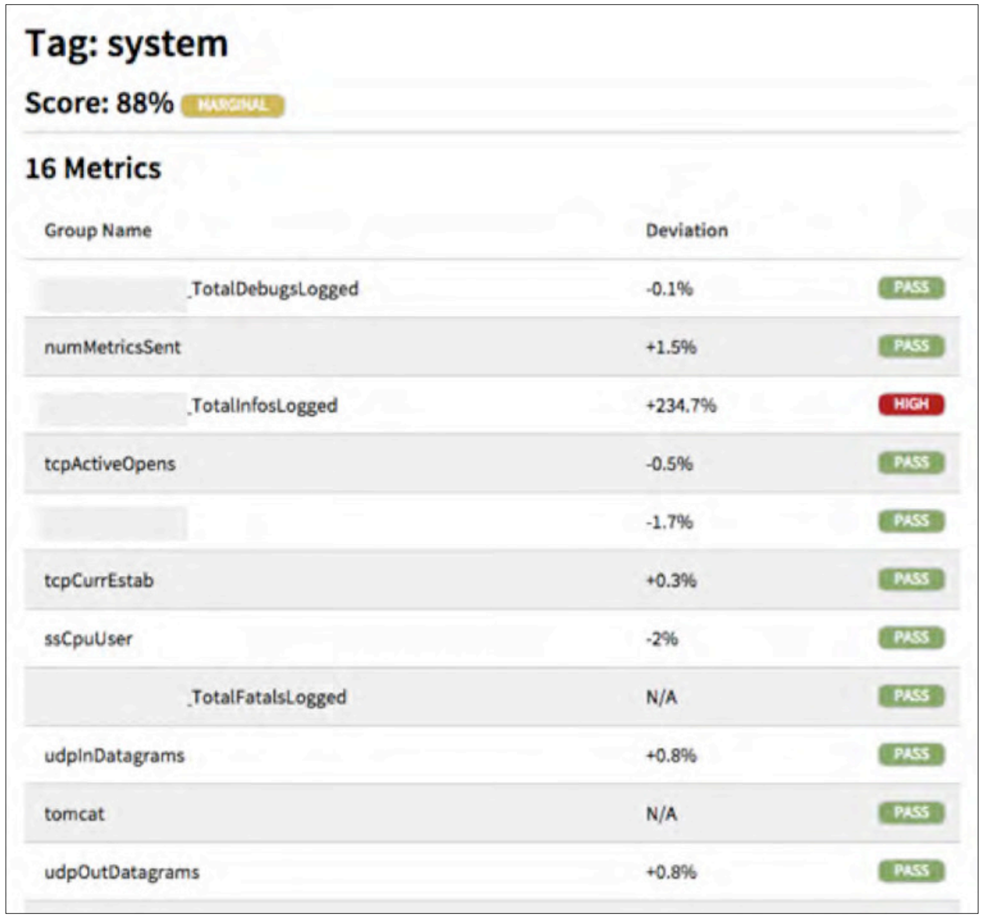


Figure 4. Canary Analysis Report Details

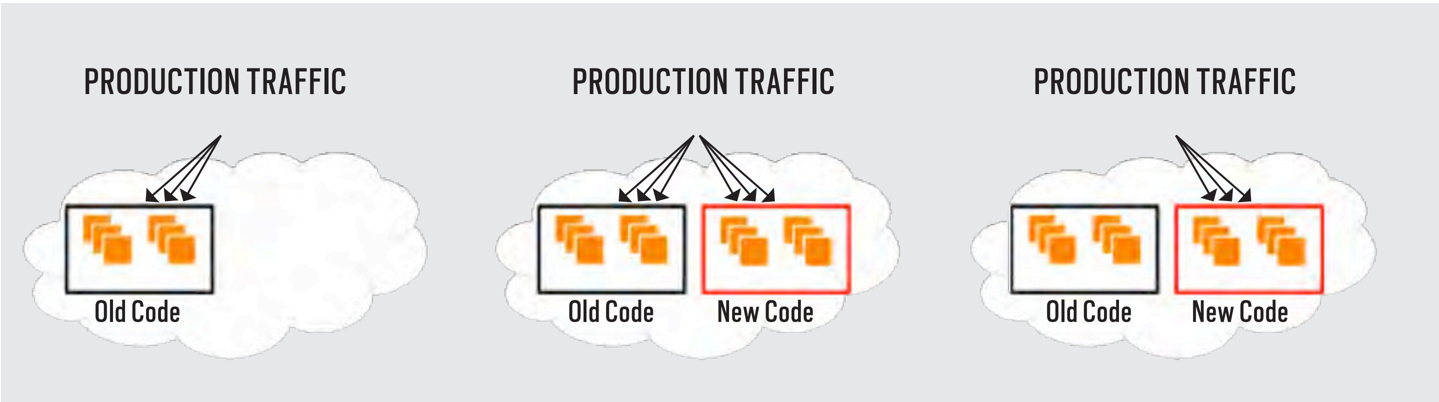


Figure 5. Red/Black Deployment

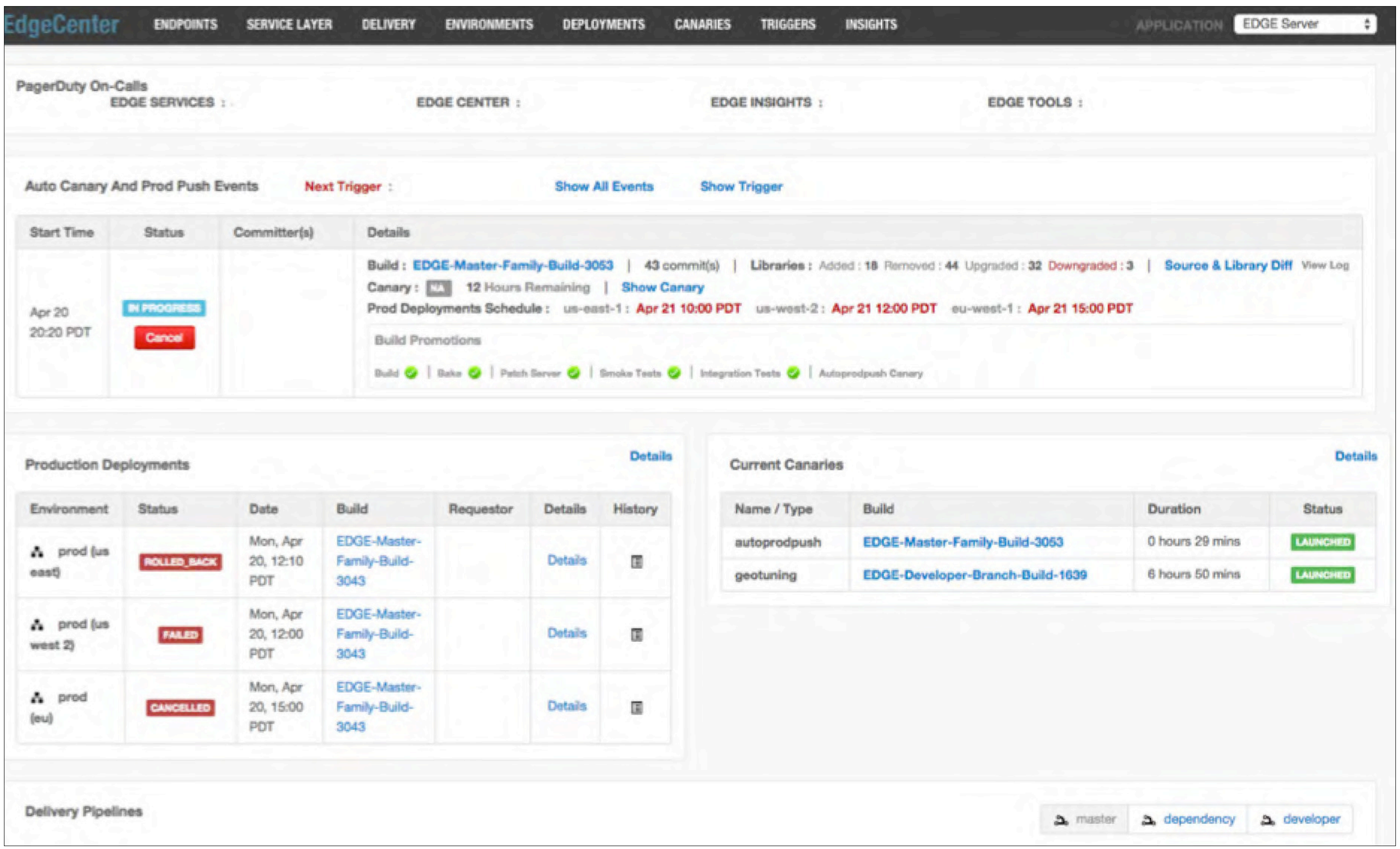


Figure 6. Delivery Insights Dashboard

tion are published to an internal event tracking system for auditability.

INSIGHTS

Microservices and CD pave the way for a distributed model for operations. Netflix engineering follows the "Operate What You Build" model. API developers are responsible for the development, testing, and operations of their code. They engage with business users and manage the rollout of the features they own. Additionally, they manage canary and alert configurations and decisions related to deployment schedules and rollbacks.

Powerful tools and rich visualizations equip developers with relevant information in a timely manner, providing detailed insights into code moving through the delivery pipeline as well as the health of the system in production. This section lists examples that are an integral part of the API delivery and operational process.

Delivery insights. The Delivery Insights tool is designed to provide the current status of all API environments, most-recent and scheduled deployments, the contents of each deployment, as well as the ability to track the flow of a given commit through the delivery pipeline (see

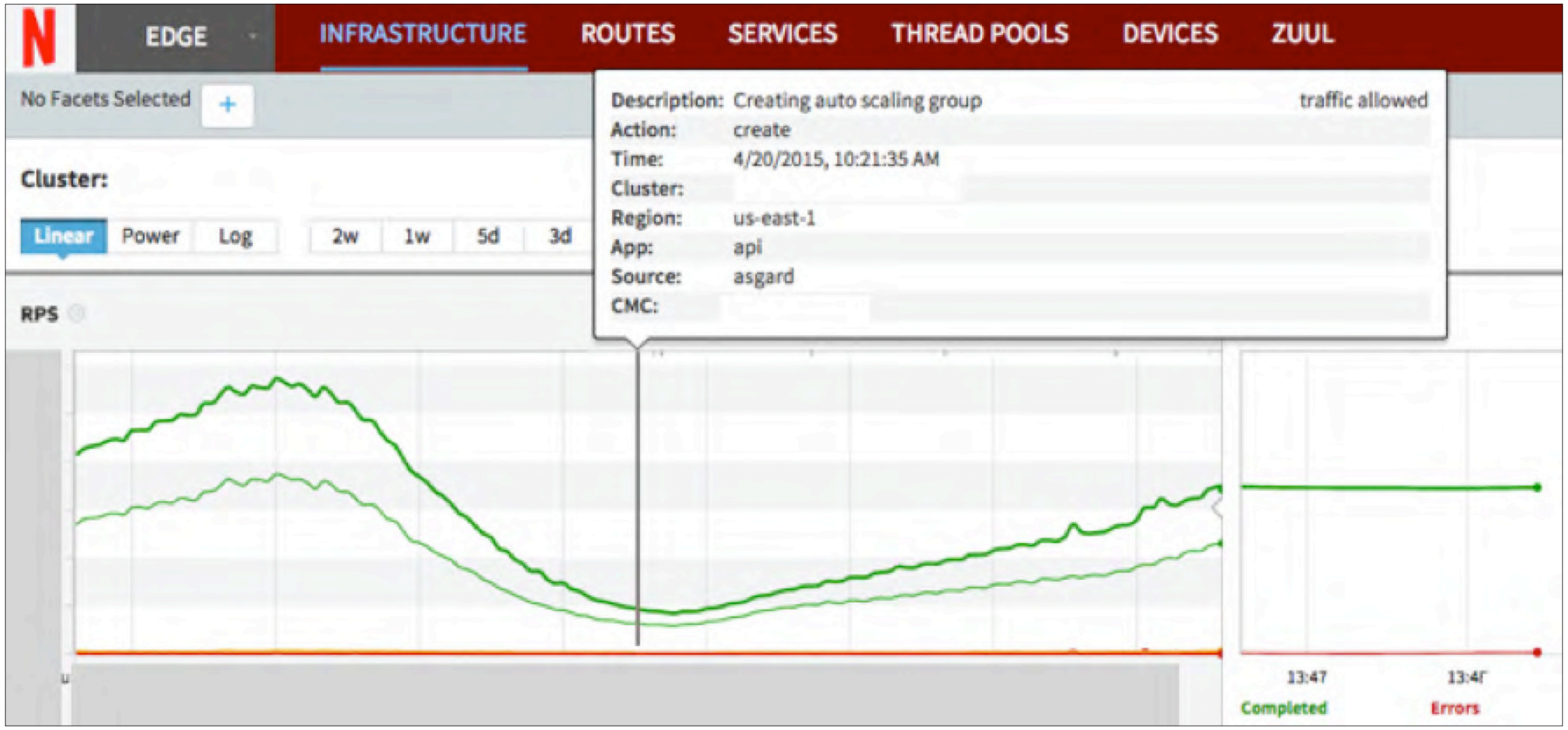


Figure 7. API Operational Health Dashboard

Figure 6). In addition, developers can follow the progress of their canary builds and initiate on-demand deployments and rollbacks. The next generation of this tool is under active development and was featured in the March/April 2015 issue of *Java Magazine*.

System usage and operational insights. Tracking production usage of a system yields valuable data that can be used to manage its lifecycle. The API service is instrumented at different levels to capture such data. One example is the use of Java agents to track calls to deprecated methods in production. This data is used to determine when it is safe to delete a method.

Another use case is for profiling code execution. Here a Java proxy injects bytecode into key request execution paths. Data is captured for each instrumented method and used to analyze behavior changes such as variations in call patterns.

Monitoring and alerting are the most-important aspects of operating large-scale distributed systems. Operational insight tools and dashboards need to be able to visualize massive amounts of data with relevant context. Dynamic visualizations and near-real-time data are invaluable when trying to zero in on the source of a problem.

An API operational health dashboard is also generated (see **Figure 7**). It shows a deployment event overlaid with the overall throughput trend for the API. The on-call engineers can go to this view regarding an anomaly if an alert is received, and check whether the anomaly correlated to a deployment.

This API dashboard is a sophisticated application with a Java back end and a front end written using Ember.js, D3, and RxJS. The back end connects to a stream processing system to receive data from thousands of API servers and streams it over WebSockets to the front end. These technologies enable complex data visualizations at scale in real time and empower developers to identify system faults and identify opportunities for improvement.

CONCLUSION

A microservices-based architecture and a heavy emphasis on automation and visualization enable Netflix to move fast while continuing to scale the service, without compromising the customer experience. To learn more about many of the tools and libraries referenced here, visit the Netflix Tech Blog or the OSS Center.

Sangeeta Narayanan is an engineering manager at Netflix. Her team is responsible for software that supports Netflix' large-scale, internet-facing services running on the Amazon cloud.

Managing Lightweight Containers with Docker, Puppet, and Vagrant

How to build, use, and orchestrate Docker containers in DevOps

MICHAEL HÜTTERMANN

BIO

In this article, I introduce Docker, the lightweight virtualization containers that have become increasingly popular in continuous delivery (CD) and DevOps pipelines. I'll explain how to integrate Docker with the infrastructure-definition tool Puppet, build Puppet-enabled Docker images, and create containers that leverage your Puppet modules. I'll integrate the tool chain with Vagrant and further streamline the CD platform. After reading this article, you will know how to use these tools to improve your development and delivery processes.

What Is Docker?

Docker is an open source framework that automates deployment and provisioning, and simplifies distribution of applications in lightweight and portable containers. These containers can then be run in a wide variety of places including on your laptop, in the cloud, or on a virtual machine (VM). Docker containers are created from Docker images, which in turn are built from Dockerfiles. Docker images are layered, and you can easily create new images based on other images. Docker images are stored in Docker registries.

Docker helps in the following ways to improve CD and DevOps implementations:

> **Vagrant is a free tool** for dynamically provisioning and managing VMs.

- Enabling productive workspaces. You can create multiple, repeatable containers for testing on your local machine, providing for distribution and sharing with your team.
- Improving the integration flow. A continuous integration pipeline can be streamlined through the use of Dockerized build slaves.
- Fostering delivery pipelines. For dynamic, repeatable integration to delivery pipelines, you can create Docker images for middleware as part of your pipeline and orchestrate them with freshly built business applications.
- Limiting variation. Infrastructure and business logic can both be included in the Docker container. Due to this converged isolation, Java ubiquity becomes even more real because you can just ship the Java runtime, middleware, and configurations—not just the standard deployment unit (typically JAR, WAR, and EAR files).
- Improving the mean time to repair/restore (MTTR). Docker containers become visual in the pipeline. This increased visibility can help an organization isolate, discover, and determine proper ownership faster.
- Integration with many tools, such as Jenkins and Maven, and enterprise repository managers, such as JFrog's Artifactory.

Containers Versus VMs

Traditional VMs are a great way to reduce physical hardware overhead. Nowadays, it is rarely necessary to set up physical hardware at the start of a project. It is pretty much standard to just scale out dynamically and to add further resources, in the form of more VMs. But VMs are considered to have a pretty big

footprint, particularly from a developer's perspective. Developers need to code/build/test every few minutes and won't like the repeated virtualization overhead. Let's now zoom in on what the overhead consists of and see how Docker can help.

In a VM, each virtualized application includes not only the application—which might be only a few megabytes, and the necessary binaries, libraries, and packages—but also an entire guest operating system, which might be multiple gigabytes, resulting in a comprehensive stack (see **Figure 1**).

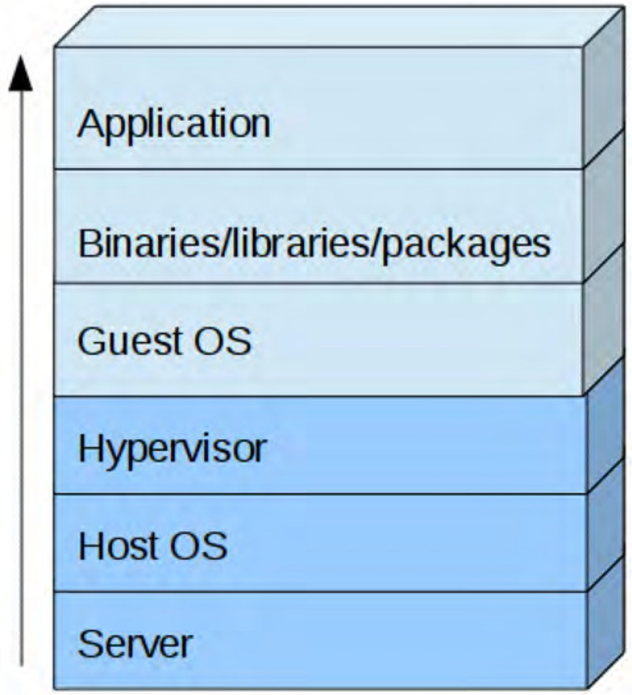


Figure 1. A typical VM stack

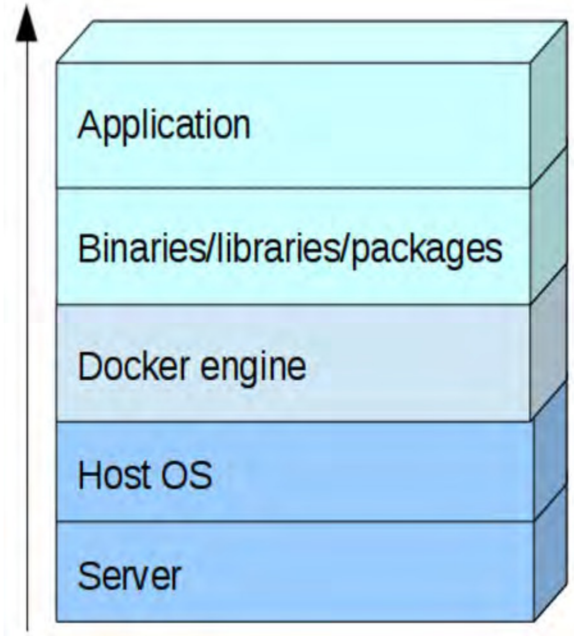


Figure 2. A typical Docker stack

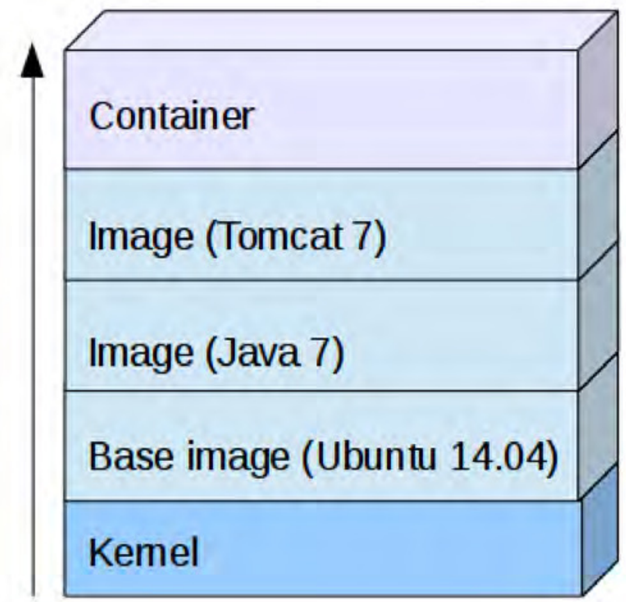


Figure 3. The example stack

The Docker container mechanism is different. It comprises just the application and its dependencies. It runs as an isolated process in user space on the host operating system, sharing the kernel with other containers. Thus, it enjoys the resource isolation and allocation benefits of VMs but has a much smaller footprint. See **Figure 2** for an overview of what the Docker stack looks like—there is no hypervisor and there is no guest OS.

The Docker engine is based on Linux techniques such as kernel cgroups and namespace isolation via Linux containers (lxc).

Building a Docker Stack

Due to its lightness, Docker facilitates new behaviors for hosts, applications, and services—especially, short-lived, disposable, single services provided in a container. What was called *service-oriented architecture* (SOA) some years ago is nowadays called *microservices*.

Let's now look at a Docker use case where we slice different services, which you can also call layers or Docker containers. In practice, it is often a good choice to split containers across architectural layers or according to the frequency of change, for example, having a single Docker image for pieces that seldom change (such as middleware) and having dedicated Docker images for pieces that often change (such as business applications).

My example consists of Java 7 running on a base image of Ubuntu Linux version 14.04. Because this is supposed to be a reusable image with infrastructure components, I call it the `infra` image. On top of it, I place an image running middleware—in our case, Tomcat 7 (see **Figure 3**). A business application will later be installed on this.

How is this done? As I mentioned earlier, Docker images are built from Docker definition files, named Dockerfiles, which are plain-text files located in dedicated directories. Let's have a look at the following Dockerfile:

Listing 1. (Indented lines are continuations of the previous line.)

```
FROM ubuntu:14.04
MAINTAINER Michael Huettermann
# Update Ubuntu
RUN apt-get update && apt-get -y upgrade
# Add oracle java 7 repository
RUN apt-get -y install software-properties-common
RUN add-apt-repository ppa:webupd8team/java
RUN apt-get -y update
# Accept the Oracle Java license
```

```
RUN echo "oracle-java7-installer
   shared/accepted-oracle-license-v1-1 boolean true" |
   debconf-set-selections
# Install Oracle Java
RUN apt-get -y install oracle-java7-installer
```

In **Listing 1**, I define a Docker image that is derived from the base image (ubuntu:14.04). There are many free Docker images available that can serve as a basis for your own Docker activities. Consult the Docker hub registry for more information.

After the meta tag about the author of this image (`MAINTAINER`), a sequence of `RUN` statements follows. Each of them expresses a shell statement applied to the target system. This image has Java 7 installed and will be able to serve as a parent image for later images. I call this image `infra`.

The next image is based on the parent image (`infra`) and runs Tomcat 7. Additionally, it carries out the business application in the form of a standard Java deployment unit (here, a WAR file). Look at the corresponding Dockerfile:

Listing 2. (Indented lines are continuations of the previous line.)

```
FROM infra
MAINTAINER Michael Huettermann
# Install curl
RUN apt-get -y install curl
# Install tomcat
RUN apt-get -y install tomcat7
RUN echo "JAVA_HOME=/usr/lib/jvm/java-7-oracle"
   >> /etc/default/tomcat7
ADD run.sh /root/run.sh
RUN chmod +x /root/run.sh
RUN curl -o /var/lib/tomcat7/webapps/all.war
    http://dl.bintray.com/mh/meow/all-1.0.0-GA.war
EXPOSE 8080
CMD ["/root/run.sh"]
```

Let's quickly walk through **Listing 2**. I define our parent image and install cURL to easily handle file downloads. Afterward, I install Tomcat 7. After adding (that is, copying) a shell script from the host to the image (that is, the `ADD` statement), I install a web application by downloading a WAR file from a remote source (Bintray) and dropping it into the deployment directly of Tomcat. I also expose ports (in this case, port 8080 is used in the container and can now be accessed by the host system or other containers). I close the Dockerfile definition by calling a `CMD` statement, which is a command executed when a container is created and run from the image.

Another use case for Vagrant is to simplify the use of Docker by using Vagrant as a wrapper for Docker.

Let's now dive deeper into the referenced shell script: run.sh.

```
#!/bin/bash
/etc/init.d/tomcat7 start
# The container will run as long as the script is running,
# which is why we need something long-lived here
exec tail -f /var/log/tomcat7/catalina.out
```

The script is just a helper script. In short, Docker containers run as long as a process runs inside the container. By running this shell script, I simply start Tomcat 7 and follow its log file: catalina.out. The latter is an unending activity, so the container stays alive. This script is called when the container is started.

Let's now build the first image (all on one line):

```
mh@saturn:~/work/ws-git/sandbox/docker/Infra$
     docker build -t infra .
```

I use the Docker client, `docker`, with parameter `build` for building the image, which will then be named `infra`. (**Note:** In this article I am using Docker 1.6.2, Puppet 3.4.3, and Vagrant 1.7.2.) This name is a tag we can use to later reference the Docker image. The Dockerfile is in our current directory, which we reference with the period at the end of the command.

The resulting console output from the build execution (see **Listing 3**) shows the execution of the individual steps of our Dockerfile. Docker will cache information once a build is performed on it. This means that Docker does not always have to process each single step again, which makes it even more lightweight. The output ends with the image ID—in our case,

56763a4909fd. The short output is a 12-character version of the long 64-digit hexadecimal string (internally, a 256-bit value). We can reference and work on images with the help of this image ID.

Listing 3. (Indented lines are continuations of the previous line.)

```
Step 0 : FROM ubuntu:14.04
 ---> 96864a7d2df3
Step 1 : MAINTAINER Michael Huettermann
 ---> Using cache
 ---> 3301dda0a0da
Step 2 : RUN apt-get update && apt-get -y upgrade
 ---> Using cache
 ---> 7a12064df01f
Step 3 : RUN apt-get -y install
     software-properties-common
 ---> Using cache
 ---> 8933d7229655
Step 4 : RUN add-apt-repository ppa:webupd8team/java
 ---> Using cache
 ---> ea29295e48f5
Step 5 : RUN apt-get -y update
 ---> Using cache
 ---> f5925321a8b4
Step 6 : RUN echo "oracle-java7-installer
    shared/accepted-oracle-license-v1-1 boolean true" |
    debconf-set-selections
 ---> Using cache
 ---> eaa1489a641f
Step 7 : RUN apt-get -y install oracle-java7-installer
 ---> Using cache
 ---> 56763a4909fd
Successfully built 56763a4909fd
```

In our example, we now build the Tomcat 7 image, which is, as you remember, based on our first image:

```
mh@saturn:~/work/ws-git/sandbox/docker/Tomcat7$
    docker build -t tomcat7 .
```

We navigate to the directory where the other Dockerfile is located. The resulting output is similar to the one above but, of course, aligned with the specific steps in that particular Dockerfile, and it closes with an individual image ID—in our case, 84b753ed5e9d.

If I type in the `docker images` command to list the available images, I see that tomcat7 is there with a virtual size of 801.9 megabytes and the image ID just discussed.

Docker's lightweight containers are faster compared with classic VMs and have become popular among developers.

Running the Image

Let's now move forward from the static view (the images) to the dynamic view (the containers). The `docker run` command runs containers from images:

```
docker run -p 8080:8080 -d tomcat7
```

With this command, I created a container from our `tomcat7` container. We run Docker as a daemon (`-d`) and map ports between the host and the guest system (`-p`). I get a container identifier afterward, if everything went well. If I check afterward to see whether the newly created container runs, by using `docker ps`, I'm told it's running and that 8080 port has been mapped.

Listing all active Docker processes shows the newly created Docker container. It is surely out of scope here to go through the complete powerful Docker API, but keep in mind that you can combine a lot of shell commands in a Docker context. For example, use `docker stop $(docker ps -a -q)` to stop all containers and `docker rm $(docker ps -a -q)` to remove all containers.

The result of our Docker run is a fully functional Docker container running Java 7, Tomcat 7, and a deployed WAR file.

Docker and Puppet

Now that we've set up the Docker container with classic Dockerfiles, you are now well prepared to learn more about how to provision Docker contain-

ers with Puppet. On the other hand, Docker itself still needs to be installed, managed, and deployed on a host. That host also needs to be managed. In turn, Docker containers might need to be orchestrated, managed, and deployed, often in conjunction with external services and tools.

The use of configuration management tools, such as Puppet, Chef [*See the related article on Chef at* *page 35*. *—Ed.*], and Ansible, is mainstream nowadays, and they help provision target systems—including middleware, services, and business applications—in a repeatable and reliable manner.

As a result of these diverse management needs, combined with the need to manage Docker itself, I think we'll see both Docker and configuration management tools being deployed in many organizations. Indeed, there is often potential for some powerful tool chains combining containers, configuration management, continuous integration, continuous delivery, and service orchestration.

Puppet has a broad audience, because it has been on the market for years, and it enables system administrators and developers to describe the target behavior of systems in a declarative way.

Our Example, with Puppet and Tomcat 7

Let's return to our example by setting up Tomcat 7. How does that look when using Puppet? **Figure 4** shows what the stack looks like now.

We now proceed with the Dockerfile:

```
FROM infra
MAINTAINER Michael Huettermann

RUN apt-get -y update
RUN apt-get -y install puppet librarian-puppet

RUN puppet module install puppetlabs-stdlib
RUN puppet module install puppetlabs-tomcat
RUN puppet module install puppetlabs-java

ADD site.pp /root/site.pp
RUN chmod +x /root/site.pp
ADD run.sh /root/run.sh
RUN chmod +x /root/run.sh

EXPOSE 8080

CMD ["/root/run.sh"]
```

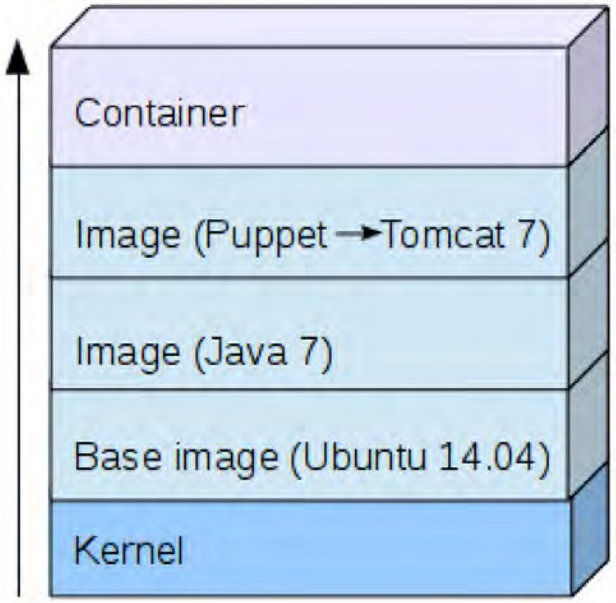


Figure 4. Our stack at present

Again, we derive our image from `infra`. After some housekeeping, in particular, updating the system with new versions and installing Librarian-puppet for seamlessly installing Puppet modules from different input channels, we install some Puppet modules. Puppet, which is known for its modularity, offers many free modules to plug in for direct use. Pay special attention here to the Puppet Tomcat module, which enables the installation and configuration of Tomcat. Our Dockerfile also embeds a Puppet manifest, named `site.pp`, which is shown in **Listing 4**:

Listing 4.

```
class { 'tomcat': } ->
class { 'java': } ->
tomcat::instance { 'install':
  source_url => 'http://mirrors.ae-online.de/apache/tomcat/
     tomcat-7/v7.0.62/bin/apache-tomcat-7.0.62.tar.gz'
} ->
exec { 'run':
  command => '/opt/apache-tomcat/bin/catalina.sh start'
} ->
tomcat::war { 'all.war':
    catalina_base => '/opt/apache-tomcat',
    war_source =>
       'http://dl.bintray.com/mh/meow/all-1.0.0-GA.war',
}
```

It is out of scope here to introduce the nuts and bolts of Puppet manifests (see the Puppet documentation for that), but thanks to Puppet's declarative

way of describing the target system's state, we get a direct impression of what's going on: We install Tomcat version 7 and afterward start it. Then, we deploy the WAR file.

Take special note of the chaining arrows, `->`, in **Listing 4**. Written with a hyphen and a greater-than sign, these ordering arrows cause the resource on the left to be applied before the resource on the right. Using this notation here is important, because we can deploy the WAR file only when the Tomcat is started, which in turn is possible only after Tomcat has been installed. Also, keep in mind that this is for demonstration purposes only. In a real-world project, for example, you would probably want to work on the Tomcat instance as a service.

Vagrant is a level above Docker in terms of abstraction, so they are not really comparable in most cases.

Now let's move forward to the run.sh script:

```
#!/bin/bash
puppet apply /root/site.pp

# The container will run as long as the script is running,
# i.e. starting a long-living process:
exec tail -f /opt/apache-tomcat/logs/catalina.out
```

The main purpose of this script is twofold. One is that we call Puppet (just locally, without a Puppet Master). The other is that we again start our long-running process—in this case, again showing the Tomcat log, which comes in handy if you follow the Docker log of the container with the command `docker logs -f [container-id]`.

Building the Docker image (with `docker build -t tomcat7puppet .`) results in a new image, which can be started with `docker run -p 8080:8080 -it tomcat7puppet`. In this case, we're running Docker interactively. As part of the processing, the output directly shows that Puppet installs Tomcat. Now, let's see how Vagrant can add value to the system.

Vagrant and Docker

Vagrant is a free tool for dynamically provisioning and managing VMs. Due to the existence of Docker, Vagrant can now be used with a Docker *provisioner* and a Docker *provider*. The Docker provisioner can automatically install Docker, pull Docker containers, and configure Docker containers.

While Vagrant is known for its support for Oracle VM VirtualBox, Vagrant has the ability to manage other types of machines as well. This is done by using other providers with Vagrant. Newer versions of Vagrant ship with support for Docker as a provider, which enables development environments to be backed by Docker containers rather than by VMs. Additionally, Vagrant provides a good workflow for developing Dockerfiles.

Vagrant is based on Vagrantfiles. The following Vagrantfile defines Docker as a provider referencing the `tomcat7puppet` image.

```
Vagrant.configure("2") do |config|
  config.vm.provider "docker" do |d|
    d.image = "tomcat7puppet"
    d.ports = [ "8080:8080" ]
  end
end
```

If I now start Vagrant using that Vagrantfile, I get the following:

```
$ vagrant up --provider=docker
Bringing machine 'default' up with 'docker' provider...
==> default: Creating the container...
    default:   Name: Vagrant_default_1432553691
    default:  Image: tomcat7puppet
    default: Volume: /home/mh/vagrant:/vagrant
    default:   Port: 8080:8080
    default:
    default: Container created: 92a523444abbd3c4
==> default: Starting container...
```

Vagrant has some important advantages when used with Docker. One important use case for Vagrant with Docker is to simplify the use of Docker by using Vagrant as a wrapper for Docker, particularly while orchestrating and managing multiple Docker containers. (There are many ways to deal with multiple containers, including Vagrant, Docker Compose, Docker Swarm, Kubernetes, and fabric8.)

In the Dockerfile in **Listing 5**, for example, I start in parallel the two con-

tainers we worked on before: one that starts Tomcat and one that starts Tomcat with Puppet. To allow the host system to access both running Tomcat instances (inside the two containers, both are running on 8080), I forward the ports accordingly.

Listing 5.

```
Vagrant.configure("2") do |config|
  config.vm.define "v1" do |a|
    a.vm.provider "docker" do |d|
      d.image = "tomcat7"
      d.ports = ["8081:8080"]
      d.name = "tomcat"
    end
  end
  config.vm.define "v2" do |a|
    a.vm.provider "docker" do |d|
      d.image = "tomcat7puppet"
      d.ports = ["8082:8080"]
      d.name = "puppet"
    end
  end
end
```

That's about as straightforward an orchestration as you can find!

On the Way to Production-Like Containers

Vagrant can serve as a Docker wrapper on systems that support Docker natively. On systems that don't, Vagrant spins up a "host VM" to run containers. You don't have to determine whether Docker is supported natively: the same configuration will work on every operating system. For that, boot2docker is used, which is a VirtualBox image of Tiny Core Linux. But this tiny image seldomly reflects a production-like setup. Thus you may want to replace it with a full-fledged Linux distribution, and provision that with Docker. See **Listing 6**, where I define a Vagrant provider, which is VirtualBox with Ubuntu Trusty64 (to set up the production-like VM), and Vagrant provisioners: shell (for bootstrapping), Puppet (for setting up infrastructure), and Docker (for setting up middleware and the business application). Crisp, isn't it?

Listing 6.

```
Vagrant.configure(2) do |config|
  config.vm.define "with-docker"
  config.vm.provider "virtualbox" do |v|
     v.name = "_withDocker"
  end
  config.vm.box = "ubuntu/trusty64"
  config.vm.box_check_update = false
  config.vm.provision :shell, path: "bootstrap.sh"
  config.vm.provision "puppet" do |puppet|
    puppet.manifests_path = "manifests"
    puppet.manifest_file = "default.pp"
  end
  config.vm.provision "docker" do |d|
     d.build_image "/vagrant/docker/Tomcat7PuppetFull",
         args: "-t tomcat7dockerfromvagrant"
     d.run "tomcat7dockerfromvagrant",
         args: "-p 8080:8080"
  end
```

Conclusion

Docker's lightweight containers are faster compared with classic VMs and have become popular among developers and as part of CD and DevOps initiatives. If your purpose is isolation, Docker is an excellent choice.

Vagrant is a VM manager that enables you to script configurations of individual VMs as well as do the provisioning. However, it is still a VM dependent on VirtualBox (or another VM manager) with relatively large overhead. It requires you to have a hard drive file that can be huge, it takes a lot of RAM, and performance can be suboptimal. Docker uses kernel cgroups and namespace isolation via lxc. This means that you are using the same kernel as the host and the same file system. Vagrant is a level above Docker in terms of abstraction, so they are not really comparable.

Configuration management tools such as Puppet are widely used for provisioning target environments. Reusing existing Puppet-based solutions is easy with Docker. You can also slice your solution, so the infrastructure is provisioned with Puppet; the middleware, the business application itself, or both are provisioned with Docker; and Docker is wrapped by Vagrant. With this range of tools, you can do what's best for your scenario. </article>

PUTTING THE CONTINUOUS IN CONTINUOUS DELIVERY

Automating the creation of deployment environments with Chef enables true continuous delivery.

BY ALAN FRYER

Continuous integration (CI) is a development practice that requires developers to integrate code into a shared repository on a regular basis, typically several times a day. Each check-in is then verified by an automated build, allowing teams to detect and fix problems early. As part of the automated builds, code-quality checks, unit tests, and integration tests will be run—and if they pass, the built artifacts will be uploaded to a central repository making them available for deployment.

Continuous delivery (CD) goes one step further than CI and focuses on automating all the build and deployment steps up to and including the user acceptance testing (UAT) environments. Most organizations have a well-established CI process, but they lack the tools to automate deployment.

This article demonstrates how to implement CD using open source tools such as Git, Maven, Nexus, Jenkins, and Private Chef. In it, I explain how to deploy a web application to a JBoss AS 6 EAP standalone instance. The web application reads and displays data from a Postgres database and is built and packaged into a WAR file and uploaded to the Nexus repository using Maven.

Note: The full listings for the code for this article can be downloaded from the listings site.

INFRASTRUCTURE MODELED AS CODE

The first step is modeling infrastructure with code, which I will do using the open source tool, Chef. This step will enable much of the automation of the other tools. In this article, I'll be modeling the configuration illustrated in **Figure 1**. As you can see, the elements I just described are present, along with a Chef server, which I'll describe in a moment. The additional machines, on which a Chef client is installed, are referred to as *nodes*.

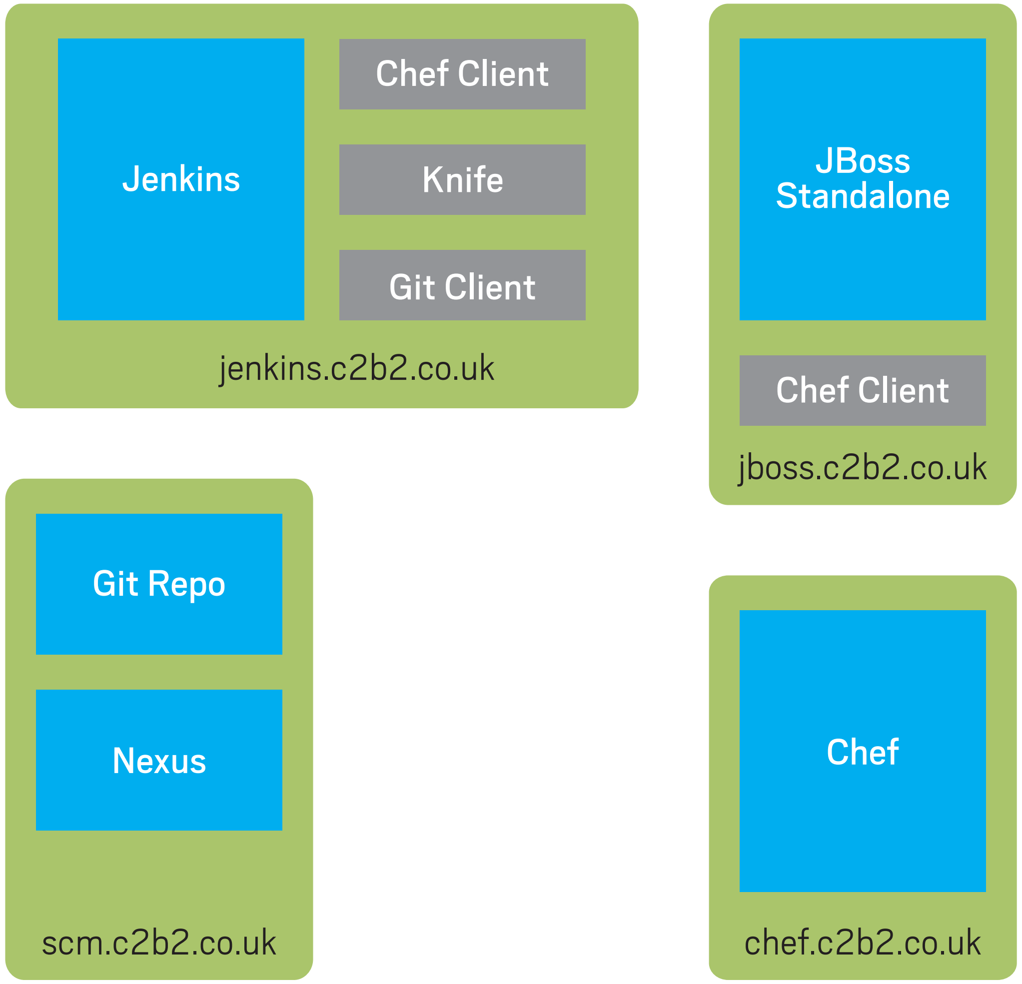


Figure 1. A typical configuration of systems using Chef

The Chef server stores all the configuration management data as attributes in cookbooks, environments, roles, and data bags.

Chef uses a DSL or Ruby code in recipes. These in turn rely on configuration information. In general, there are four primary elements of interest:

- **Cookbooks** contain Chef DSL/Ruby code in recipes, libraries, and definitions of what resources/operations need to be carried out on a node. The Ruby code (that is, the logic) references attributes defined in the cookbook, roles, data bags, or environments.
- **Data bags** define global attributes that can be used by all code in the cookbooks.
- **Environments** are associated with nodes managed by Chef. They define the cookbooks and any attributes specific for that system. The cookbooks can be pinned to a version, allowing different versions to be applied to the DEV, TEST, and PROD environments.
- **Roles** describe the purpose of a server and define the run list and order of recipes to be applied to the node.

The Chef server stores all the configuration management data as attributes in cookbooks, environments, roles, and data bags. The cookbooks and roles typically store static attributes that don't change between environments, whereas environments are used to define environment-specific properties.

The attributes in environments, roles, and data bags can be created either using the Chef console or as source files in a JSON format.

All servers managed by Chef—that is, the nodes—require the Chef client to be installed and configured on them to connect to the Chef server over HTTPS. The node jboss.c2b2.co.uk shown in **Figure 1** is registered with the Chef server at chef.c2b2.co.uk and assigned the environment `C2B2Dev`, which is defined in **Listing 1**.

Listing 1.

```
{
 "name": "C2B2Dev",
 "description": "",
 "cookbook_versions": {
 },
 "json_class": "Chef::Environment",
 "chef_type": "environment",
 "default_attributes": {
  "jboss": {
   "jboss_home": "/opt/jboss/jboss-eap-6.3",
   "jboss_install": "/opt/jboss",
  . . .
```

It is assigned the role of `JbossStandalone` in **Listing 2**.

Listing 2.

```
{
 "name": "JbossStandalone",
 "description": "",
 "json_class": "Chef::Role",
 "default_attributes": {
 },
 "override_attributes": {
 },
 "chef_type": "role",
 "run_list": [
  "recipe[jboss6::installjboss]",
  "recipe[jboss6::adduser]",
  "recipe[jboss6::installpostgresmodule]",
  "recipe[jboss6::startstandalone]",
  "recipe[jboss6::configurestandalone]"
 ],
 "env_run_lists": {
 }
}
```

Chef provides a command line utility called `knife`, which is used to manage the Chef objects and environments. The `knife ssh` subcommand is used to invoke SSH commands (in parallel) on a subset of nodes. Which nodes it runs on can be determined by a search run on the Chef server. This capability allows you to run the Chef client on nodes with a certain

role in a target environment. For example, you can install JBoss, and then configure and deploy the web application from a single `knife` command.

```
knife ssh -x afryer
   chef_environment:C2B2Dev AND role: JbossStandalone"
   "sudo -u root chef-client -l info"
```

The `knife ssh` command queries the Chef server and returns a list of matching nodes in the `C2B2Dev` environment with the `JbossStandalone` role associated with them. An SSH session is started—as the user `afryer`—on each node returned by the search, and it executes via the Chef client with `sudo` access. The Chef client connects to the Chef server and updates the attributes in the node's data set with the values set in the environment and cookbooks assigned to the node. It then executes the recipes defined in the `C2B2Dev` role's run list.

The recipes read the respective configuration management data used to install and configure the application by referencing the attributes on the node. For example, reading the JBoss home directory into a variable would be done as follows:

```
jboss_home = node['jboss']['jboss_home']
```

where the attribute value is defined in the environment `C2B2Dev`, as shown in the last few lines of **Listing 1**, earlier.

EXAMINING CHEF CODE

In the rest of this article, I show how some of the key operations are specified. The inline listings contain relevant excerpts. For copies of the full listings, go to the download area for this issue.

Install JBoss. The recipe in **Listing 3** creates the user `jboss`, downloads the JBoss EAP 6.3.0 binaries stored as a zip file from Nexus, and unpacks them to the installation directory. The recipe uses the Chef DSL to define the Resources needed to create the user to run JBoss and the directory in which to install it. The JBoss zip file is downloaded from Nexus using the `remote_file` resource, specifying the URL to the Nexus Rest API with the arguments for the repository, group Id, artifact Id, version, and packaging. The `execute` resource is used to run a Linux command to unpack the zip file and change the owner of the JBoss directory structure to the user `jboss`.

Add user. The `adduser` recipe in **Listing 4** uses the `execute` resource to call the shell script `$JBOSS_HOME/bin/add-user.sh` to create the management user/password used to secure the JBoss Administration console. Chef only allows attributes to be stored in an encrypted form in data bags, so sensitive data such as passwords should always be stored in an encrypted data bag.

To encrypt a data bag, you first need to create a secret key to be used for encryption and decryption. To create a secret key, on the Chef workstation, execute the following command:

```
openssl rand -base64 512 >
    /home/afryer/certs/c2b2_secret_key
```

This file needs be copied to all the nodes running the Chef client and referenced in any recipes where encrypted data bags are used. To create an encrypted data bag, enter the following command:

```
knife data bag create- -secret-file
  /home/afryer/certs/c2b2_secret_key
  bancs_dev_credentials c2b2
```

The text editor associated with the `knife` tool opens a temporary JSON file in plain text for editing with the `id` attribute set to the name of the data bag item entered. You can then add the attributes to be encrypted to the file using the format `"attribute-name": "attribute-value"`, separating new attributes added with a comma.

```
{
   "id": "c2b2",
   "mgmt_pwd": "password"
}
```

We have now created a mechanism that automates the deployment of applications to multiple nodes from a single command-line call.

When the file is saved and the editor is closed, the data bag is created on the server in an encrypted format. An encrypted data bag must be created to contain credentials for each environment. The password for the management user `mgm_pwd` is read from the data bag using the recipe in **Listing 4A**.

Install Postgres module. The `installpostgresmodule` recipe in **Listing 5** installs the Postgres JDBC driver in the JBoss `modules` folder. The folder `$JBOSS_HOME/modules/postgres/main` is created using the `directory` resource and copies the cookbook file `module.xml` (**Listing 6**) to this folder using the `cookbook_file` resource.

Start JBoss Standalone instance. The `startstandalone` recipe in **Listing 7** starts the JBoss Standalone instance by using the `execute` resource to run the shell script `$JBOSS_HOME/bin/standaloe.sh` (**Listing 7A**) in nohup mode.

The management address and JBoss server is bound to the DNS name of the server by setting the options `-b` and `-Djboss.bind.address.management` with the value obtained from the node attribute `node['fqdn']`.

Configure JBoss Standalone instance. The `configurestandalone` recipe in **Listing 8** configures the Postgres datasource, downloads the application WAR file from Nexus, and deploys it to the JBoss server. The configuration and deployment are performed by executing a JBoss CLI script against the running server. The CLI script (**Listing 9**) is defined as a cookbook template, which is an embedded Ruby template that is used to generate the CLI script based on the variables and logic contained within the template. The template is generated using the following resource definition, which passes in the variables read from the node:

```
template cli_script do
  source "configure-standalone.cli.erb"
  variables(
    ds_jndi_name: "postgresDS",
    ds_pool_name: "postgresDS",
    db_host: node['jboss']['database_hostname'],
    db_port: node['jboss']['database_port'],
    db_name: node['jboss']['database_name'],
    db_user: node['jboss']['database_user'],
    db_password: creds["postgres"],
    jboss_install: jboss_install,
    artifact: "#{artifact_id}.war",
    version: version
  )
  owner process_user
  group process_group
end
```

The `execute` resource then runs the shell script `$JBOSS_HOME/bin/jboss-cli.sh` with the option `--file=configure-standalone.cli` to run the CLI script generated from the template.

INTEGRATING CHEF AND JENKINS

The Chef client and `knife` tool need to be configured on the server running Jenkins. This step requires creating the user `jenkins` on the Chef server using the management console, and linking it to an organization—in this case, C2B2. A private key for the user is generated. It is saved to the `knife` tool's configuration folder `/home/jenkins/.chef`. The file `knife.rb` configures the connection to the Chef server, together with the location of the private key used to authenticate the connection and location of the cookbook's folder in the local Chef repository (**Listing 10**).

To execute a Chef run, a Jenkins Freestyle Project is used. The build checks out the Chef code from a specific branch in a Git repository to the build's workspace, relying on `CHEF_ENVIRONMENT` and `CHEF_ROLE`. The checked-out code includes the shell script in **Listing 11**, which is executed as part of the build.

The shell script uploads the checked-out version of the cookbook from the Git repository branch configured in the build, which executes the `knife ssh` command, running the Chef client on the nodes matching the environment and the role passed in to the script.

The JBoss installer and web application are both stored in a Nexus repository and downloaded using the group id, artifact id, and version. The web application is built from a separate Jenkins project that uses Maven to build, package, and release the packaged WAR file to Nexus.

CONCLUSION

This work has generated a mechanism that automates the deployment of applications to an environment on multiple nodes ultimately from a single command-line call. By configuring the `knife` tool on the Jenkins server, a shell script can be executed to run a `knife ssh` command to initiate an automatic deployment to an environment. This can be done either automatically when a test run completes, on a schedule, or ad hoc with manual triggers, enabling continuous delivery to be achieved in the development, test, and UAT environments.

Once configured, Chef enables application deployments to be made in a repeatable manner, significantly reducing deployment times and releasing development/support teams to focus on value-added work. This approach lends itself to provisioning a consistent configuration across multiple environments, which reduces time spent trying to debug configuration errors that typically occur with manual deployments. With careful design of the recipes, solutions such as Chef can also help to promote the building of standard environments quickly for new projects using a certified middleware stack configuration. This enforces good practices across all projects, improving the quality of the systems delivered and reducing development and support costs. </article>

Alan Fryer has been designing Java EE systems for 12 years and has extensive experience in Java EE consultancy, troubleshooting, problem analysis, performance tuning, production infrastructure design, and administration. As a principal consultant at C2B2, he is currently working on a long-term assignment with a large government department, providing middleware consultancy to the DevOps teams that develop and support Java EE applications.

VENKAT
SUBRAMANIAM
BIO

Part 1

Functional Programming in Java: Using Collections

Use Java 8 lambda expressions to greatly reduce code clutter.

In this two-part article, I demonstrate how to use Java 8 lambda expressions to take advantage of the functional style of programming to create more-expressive and concise code with less mutability and fewer errors.

Using Collections

Collections of numbers, strings, and objects are used so commonly in Java that removing even a small amount of ceremony from coding them can reduce code clutter greatly. In this article, I explore the use of lambda expressions to manipulate collections. I use them to iterate collections, extract elements from them, and easily concatenate their elements.

After you read this article, your Java code to manipulate collections might never be the same—it'll be concise, expressive, elegant, and more extensible than ever before.

Iterating Through a List

Iterating through a list is a basic operation on a collection, but over the years it's gone through a few significant changes. We'll begin with the old and evolve an example—enumerating a list of names—to the elegant style.

We can easily create an immutable collection of a list of names with the following code:

```
final List<String> friends =
    Arrays.asList("Brian", "Nate", "Neal",
                  "Raju", "Sara", "Scott");
```

Here's the habitual, but not so desirable, way to iterate and print each of the elements.

```
for(int i = 0; i < friends.size(); i++) {
    System.out.println(friends.get(i));
}
```

I call this style the *self-inflicted wound pattern*—it's verbose and error prone. We have to stop and wonder, "Is it `i <` or `i <=`?" This is useful only if we need to manipulate elements at a particular index in the collection, but even then, we can opt to use a functional style that favors immutability, as we'll discuss soon.

Java also offers a construct that is a bit more civilized than the good old `for` loop.

```
for(String name : friends) {
   System.out.println(name);
}
```

Under the hood this form of iteration uses the `Iterator` interface and calls into its `hasNext()` and `next()` methods.

Both these versions are *external iterators*, which mix *how* we do it with *what* we'd like to achieve. We explicitly control the iteration with them, indicating where to start and where to end; the second version does that under the hood using the `Iterator` methods. With explicit control, the `break` and

`continue` statements can also help manage the iteration's flow of control.

The second construct has less ceremony than the first. Its style is better than the first if we don't intend to modify the collection at a particular index. Both of these styles, however, are imperative and we can dispense with them in modern Java by using a functional approach.

There are quite a few reasons to favor the change to the functional style:

- The `for` loops are inherently sequential and are quite difficult to parallelize.
- Such loops are nonpolymorphic; we get exactly what we ask for. We passed the collection to `for` instead of invoking a method (a polymorphic operation) on the collection to perform the task.
- At the design level, the code fails the "Tell, don't ask" principle. We ask for a specific iteration to be performed instead of leaving the details of the iteration to underlying libraries.

It's time to trade in the old imperative style for the more elegant functional-style version of *internal iteration*. With an internal iteration we willfully turn over most of the *hows* to the underlying library so we can focus on the essential *whats*. The underlying function will take care of managing the iteration. Let's use an internal iterator to enumerate the names.

The `Iterable` interface has been enhanced in JDK 8 with a special method named `forEach()`, which accepts a parameter of type `Consumer`. As the name indicates, an instance of `Consumer` will consume, through its `accept()` method, what's given to it. Let's use the `forEach()` method with the all-too-familiar anonymous inner class syntax.

```
friends.forEach(new Consumer<String>() {
   public void accept(final String name) {
       System.out.println(name);
   }
});
```

Here, we have invoked `forEach()` on the `friends` collection and passed an anonymous instance of `Consumer` to it. The `forEach()` method will invoke the `accept()` method of the given `Consumer` for each element in the collection and perform a specified action. In this example, it merely prints the given value, which is a name.

Let's look at the output from this version, which is the same as the output from the two previous versions:

```
Brian
Nate
Neal
Raju
Sara
Scott
```

We changed just one thing: we traded in the old `for` loop for the new internal iterator `forEach()`. As for the benefit, we went from specifying how to iterate to focusing on what we want to do for each element. The bad news is the code looks a lot more verbose—so much that it can drain away any excitement about the new style of programming. Thankfully, we can fix that quickly; this is where lambda expressions and the new compiler magic come in. Let's make one change again, replacing the anonymous inner class with a lambda expression.

The Java compiler treats single-parameter lambda expressions as special. We can leave off the parentheses around the parameter if the parameter's type can be inferred.

```
friends.forEach((final String name) ->
     System.out.println(name));
```

That's a lot better. We look at less code, but watch closely to see what's in there. The `forEach()` is a higher-order function that accepts a lambda expression or block of code to execute in the context of each element in the list. The variable `name` is bound to each element of the collection during the call. The underlying library takes control of how any lambda expressions are evaluated. It can decide to perform them lazily, in any order, and exploit parallelism as it sees fit.

This version produces the same output as the previous versions. The internal-iterator version is more concise than the other ones. In addition, when we use it we're able to focus our attention on what we want to achieve for each element rather than how to sequence through the iteration—it's declarative.

This version has a limitation, however. Once the `forEach` method starts, unlike in the other two versions, we can't break out of the iteration. (There are facilities to handle this limitation.) As a consequence, this style is useful

in the common case where we want to process each element in a collection. Later we'll see alternate functions that give us control over the path of iteration.

The standard syntax for lambda expressions expects the parameters to be enclosed in parentheses, with the type information provided and comma separated. The Java compiler also offers some lenience and can infer the types. Leaving out the type is convenient, requires less effort, and is less noisy. Here's the previous code without the type information.

```
friends.forEach((name) ->
    System.out.println(name));
```

In this case, the Java compiler determines that the `name` parameter is a String type, based on the context. It looks up the signature of the called method, `forEach()` in this example, and analyzes the functional interface it takes as a parameter. It then looks at that interface's abstract method to determine the expected number of parameters and their types. We can also use type inference if a lambda expression takes multiple parameters, but in that case we must leave out the type information for all the parameters; we have to specify the type for *none* or for *all* of the parameters in a lambda expression.

The Java compiler treats single-parameter lambda expressions as special. We can leave off the parentheses around the parameter if the parameter's type is inferred:

```
friends.forEach(name ->
    System.out.println(name));
```

There's one caveat: inferred parameters are non-`final`. In the previous example, where we explicitly specified the type, we also marked the parameter as `final`. This prevents us from modifying the parameter within the lambda expression. In general, modifying parameters is in poor taste and leads to errors, so marking them `final` is a good practice. Unfortunately, when we favor type inference we have to practice extra discipline not to modify the parameter, because the compiler will not protect us.

The map() method of a new Stream interface can help us avoid mutability and make the code concise. A Stream is much like an iterator on a collection of objects.

We have come a long way with this example and reduced the code quite a bit. But let's take one last step to tease out another ounce of conciseness:

```
friends.forEach(System.out::println);
```

In this code, we used a *method reference*. Java lets us simply replace the body of code with the method name of our choice. We will dig into this further in the next section, but for now let's reflect on the wise words of Antoine de Saint-Exupéry: "Perfection is achieved not when there is nothing more to add, but when there is nothing left to take away."

Lambda expressions helped us concisely iterate over a collection. Next we'll cover how they help remove mutability and make the code even more concise when transforming collections.

Transforming a List

Manipulating a collection to produce another result is as easy as iterating through the elements of a collection. Suppose we're asked to convert a list of names to all capital letters. Let's explore some options to achieve this.

Java's `String` is immutable, so instances can't be changed. We could create new strings in all caps and replace the appropriate elements in the collection. However, the original collection would be lost; also, if the original list is immutable, as it is when created with `Arrays.asList()`, the list can't change. It would also be hard to parallelize the computations.

Creating a new list that has the elements in all uppercase is better.

That suggestion may seem quite naive at first; performance is an obvious concern we all share. You're likely to find, however, that the functional approach often yields surprisingly better performance than the imperative approach. Let's start by creating a new collection of uppercase names from the given collection.

```
final List<String> uppercaseNames =
    new ArrayList<String>();

for(String name : friends) {
    uppercaseNames.add(name.toUpperCase());
}
```

In this imperative style, we created an empty list and then populated it with uppercase names, one element at a time, while iterating through the original list. As a first step to move toward a functional style, we could use the internal iterator `forEach()` method from the "Iterating Through a List" section, to replace the `for` loop:

```
final List<String> uppercaseNames =
    new ArrayList<String>();
friends.forEach(name ->
    uppercaseNames.add(name.toUpperCase()));
System.out.println(uppercaseNames);
```

We used the internal iterator, but that still required the empty list and the effort to add elements to it. We can do a lot better.

The `map()` method of a new `Stream` interface can help us avoid mutability and make the code concise. A `Stream` is much like an iterator on a collection of objects and provides some nice *fluent functions*. Using the methods of this interface, we can compose a sequence of calls so the code reads and flows in the same way we'd state the problem, making it easier to read.

The `Stream`'s `map()` method can map or transform an input sequence to an output sequence—that fits quite well for the task at hand.

```
friends.stream()
    .map(name -> name.toUpperCase())
    .forEach(name -> System.out.print(name + " "));
System.out.println();
```

The method `stream()` is available on all collections in JDK 8, and it wraps the collection into an instance of `Stream`. The `map()` method applies the given lambda expression or block of code within the parentheses on each element in the `Stream`. The `map()` method is quite unlike the `forEach` method, which simply runs the block in the context of each element in the collection. In addition, the `map()` method collects the result of running the lambda expression and returns the result collection. Finally, we print the elements in this result using the `forEach()` method. The names in the new collection are in all capital letters:

```
BRIAN NATE NEAL RAJU SARA SCOTT
```

The `map()` method is very useful for mapping or transforming an input collection into a new output collection. This method will ensure that the same number of elements exists in the input and the output sequence. However, element types in the input don't have to match the element types in the output collection. In this example, both the input and the output are a collection of strings. We could have passed to the `map()` method a block of code that returned, for example, the number of characters in a given name. In this case, the input would still be a sequence of strings, but the output would be a sequence of numbers, as in the next example.

```
friends.stream()
    .map(name -> name.length())
    .forEach(count -> System.out.print(count + " "));
```

The result is a count of the number of letters in each name:

```
5 4 4 4 4 5
```

The versions using the lambda expressions have no explicit mutation; they're concise. These versions also didn't need any initial empty collection or garbage variable; that variable quietly receded into the shadows of the underlying implementation.

Using Method References

We can make the code be just a bit more concise by using a feature called *method reference*. The Java compiler will take either a lambda expression or a reference to a method where an implementation of a functional interface is expected. With this feature, a short `String::toUpperCase` can replace `name ->name.toUpperCase()`, like so:

```
friends.stream()
       .map(String::toUpperCase)
       .forEach(name -> System.out.println(name));
```

Java knows to invoke the `String` class's given method `toUpperCase()` on the parameter passed in to the synthesized method—the implementation of the functional interface's `abstract` method. That parameter reference is implicit here. In simple situations such as the previous example, we

can substitute method references for lambda expressions (see **sidebar**).

In the preceding example, the method reference was for an instance method. Method references can also refer to `static` methods and methods that take parameters. We'll see examples of these later.

As we just saw, lambda expressions helped us enumerate a collection and transform it into a new collection. They can also help us concisely pick an element from a collection, as we'll see next.

Finding Elements

The now-familiar elegant methods to traverse and transform collections will not directly help pick elements from a collection. The `filter()` method is designed for that purpose.

From a list of names, let's pick the ones that start with the letter *N*. Because there may be zero matching names in the list, the result may be an empty list. Let's first code it using the old approach.

```
final List<String> startsWithN =
    new ArrayList<String>();
for(String name : friends) {
    if(name.startsWith("N")) {
    startsWithN.add(name);
    }
}
```

That's a chatty piece of code for a simple task. We created a variable and initialized it to an empty collection. Then we looped through the collection, looking for a name that starts with the desired letter. If found, we added the element to the collection.

Let's refactor this code to use the `filter()` method, and see how it changes things.

```
final List<String> startsWithN =
   friends.stream()
          .filter(name -> name.startsWith("N"))
          .collect(Collectors.toList());
```

The `filter()` method expects a lambda expression that returns a `boolean` result. If the lambda expression returns `true`, the element in context while executing that lambda expression is added to a result collection; it's skipped otherwise. Finally the method returns a stream with only elements for which the lambda expression yielded `true`. In the end, we transformed the result into a `List` using the `collect()` method.

Let's print the number of elements in the result collection.

```
System.out.println(
    String.format(
       "Found %d names", startsWithN.size()));
```

When Should You Use Method References?

I typically use lambda expressions much more than method references when programming in Java. That doesn't mean method references are unimportant or less useful, though. They are nice replacements when the lambda expressions are really short and make simple, direct calls to either an instance method or a static method. In other words, if lambda expressions merely pass their parameters through, we can replace them with method references.

These candidate lambda expressions are much like Tom Smykowski, in the movie *Office Space*, whose job is to "take specifications from the customers and bring them down to the software engineers." For this reason, I call the refactoring of lambdas to method references the *office-space pattern*.

In addition to conciseness, with method references we gain the ability to use more directly the names already chosen for these methods.

There's quite a bit of compiler magic taking place under the hood when we use method references. The method reference's target object and parameters are derived from the parameters passed to the synthesized method. This makes the code with method references much more concise than the code with lambda expressions. However, we can't use this convenience if we need to manipulate parameters before sending them as arguments or to tinker with the call's results before returning them.

From the output, it's clear that the method picked up the proper number of elements from the input collection.

```
Found 2 names
```

The `filter()` method returns an iterator just like the `map()` method does, but the similarity ends there. Whereas the `map()` method returns a collection of the same size as the input collection, the `filter()` method might not. It might yield a result collection with a number of elements ranging from zero to the maximum number of elements in the input collection. However, unlike `map()`, the elements in the result collection that `filter()` returned are a subset of the elements in the input collection.

Conclusion

The conciseness we've achieved by using lambda expressions so far is nice, but code duplication might sneak in quickly if we're not careful. I'll address this concern in the second half of this article in the upcoming issue. **</article>**

This article was adapted from Functional Programming in Java: Harnessing the Power of Java 8 Lambda Expressions *with kind permission from the publisher, The Pragmatic Bookshelf.*

LEARN MORE

- Video of the author lecturing on this topic
- What is higher-order programming?
- Why functional programming is on the rise

ANTONIO GONCALVES

BIO

Part 2

Contexts and Dependency Injection: the New Java EE Toolbox

Integrating third-party frameworks

This series of four articles attempts to demystify Contexts and Dependency Injection (CDI). In the previous article, I discussed what strong typing means in dependency injection. Here, I explain how to integrate third-party frameworks. The next article will focus on how to achieve loose coupling with interceptors, decorators, and events. The final article will cover the integration of CDI within Java EE.

CDI 1.0 first appeared in Java EE 6. Since then, not only has it been updated with each platform release, but it has become a central piece of Java EE: most of the specifications use it internally. CDI can also integrate third-party frameworks to bring coherence and extensibility to the platform.

In the previous article, I examined how CDI can inject beans into other beans. The problem we now face is that when we want to integrate third-party frameworks, their classes are not discovered by CDI and so they can't be injected. Thanks to producers, however, CDI enables us to turn any Java class into a CDI bean so it can be managed by the CDI container. Once a class is managed, we can use qualifiers and alternatives as we saw in the previous article. Thanks to disposers, a produced bean can be cleaned up easily.

Bean Discovery

When an application starts, the CDI container performs *bean discovery*. This is the process in which the container searches for CDI beans in all bean archives that are declared in the application classpath. During bean discovery, the CDI container detects definition errors and deployment problems. If any exception is raised, the deployment is canceled and the application is unavailable. If no exception is raised, all the CDI beans are discovered, injection points are referenced, and the application begins running. But not every Java class is a CDI bean. Bean discovery is done only in bean archives. Other archives are not treated by CDI.

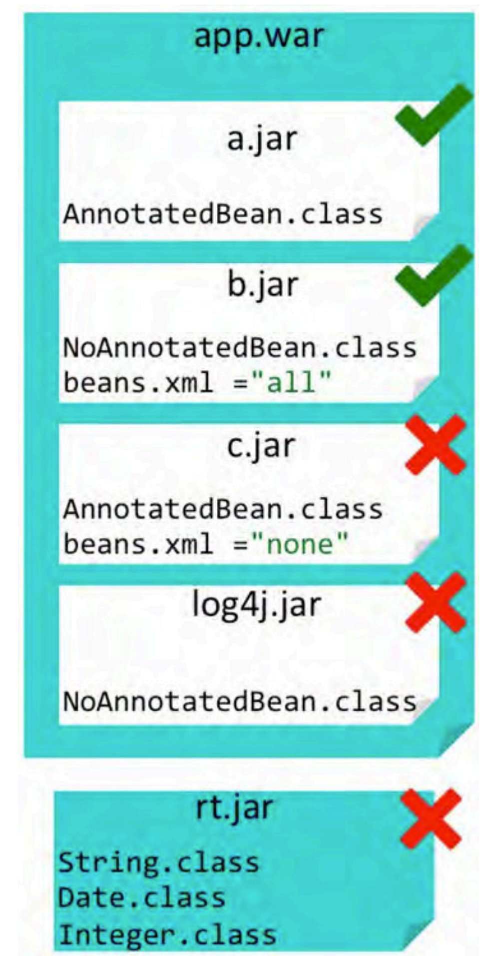


Figure 1. What constitutes a bean archive

CDI Container and CDI Beans

The *CDI container* is a runtime environment that manages CDI beans and provides them with certain services. These services could be lifecycle management, dependency injection, interception, decoration, event handling, and so on.

When we talk about a *CDI bean*, we simply mean a Java class that follows certain patterns and is managed by a CDI container. A CDI bean must be a concrete class with a default constructor; it can have

optional qualifiers, scopes, an expression language name, or a set of interceptor bindings. In fact, with very few exceptions, potentially every Java class that has a default constructor can be a CDI bean, if packaged in a bean archive.

Bean Archives

CDI doesn't define any particular deployment archive. We can package CDI beans in JARs or WARs. However, CDI discovers only beans that are packaged following certain rules—those in a *bean archive*. A bean archive is any archive that packages Java classes annotated with CDI annotations. For example, if a class contains a CDI annotation (qualifier, interceptor binding, and so on), that class is said to be an `AnnotatedBean`; if it appears in `a.jar`, then `a.jar` is said to be a bean archive.

The other possibility is to use a `beans.xml` deployment descriptor in a JAR with a bean discovery mode set to `all`. In **Figure 1**, `b.jar` packages no annotated beans, but because it contains a `beans.xml` file, it is a bean archive.

Any other packaging is said to be a plain archive. For example, if the bean discovery mode in `beans.xml` is set to `none` (meaning that we tell CDI not to discover any beans in this archive), then `c.jar` is not treated as a bean archive. This means that even if `c.jar` packages CDI beans, they will not be discovered and not managed by CDI. The other common case is when the archive does not contain any annotated beans and has no `beans.xml` descriptor. In **Figure 1**, `log4j.jar` and `rt.jar` do not contain any of those, so they are not bean archives.

When we package a web application in a WAR file, it is common to bundle third-party frameworks that are not bean archives. For example, the `app.war` web application needs Log4j, a third-party logging framework, and of course, Java APIs such as `String`, `Date`, and `Integer`, which are packaged in the `rt.jar` file. This `rt.jar` file contains the Java runtime environment (JRE) classes and doesn't have a `beans.xml` deployment descriptor. So, these classes cannot be discovered or managed by CDI, and therefore they cannot be injected. To be injected, they first need to be *produced*.

Producers

Basically, a *producer* provides CDI capabilities to a plain old Java object (POJO) or to any datatype. Why would we want to do that? The answer is that there are plenty of cases where we need additional control. What if we need to decide at runtime which implementation of a datatype or logger to instantiate and inject?

That's where producers are beneficial: they enable third-party frameworks to be used with CDI by exposing their objects as CDI beans. To do this, declare a field or a method to be produced by annotating it with `@Produces`.

Producer fields. Let's look at an example that illustrates producers. In **Listing 1**, the `BookService` class has a method to create a book object given a title. This method also generates an ISBN number by concatenating a prefix, a random number, and a postfix. These attributes are part of the service and, as you can see, are not initialized.

Listing 1.

```
public class BookService {

    @Inject
    private String prefix;
    @Inject
    private int random;
    @Inject
    private long postfix;

    public Book createBook(String title) {
        String number = prefix + "-" +
            random + "-" + postfix;
        return new Book(title, number);
    }
}
```

The goal is to ask CDI to inject the values of these attributes. As mentioned earlier, `String`, `int`, and primitive datatypes are part of the JRE, and therefore cannot be injected. The only way for this to work is to produce them.

To do this, we just take any class in our application, such as `NumberProducer` (**Listing 2**), but it could have been called anything else. We declare some attributes, initialize them, and ask CDI to produce them with the `@Produces` annotation. This means that all the produced datatypes can now be injected with `@Inject` anywhere in our application. Notice that CDI does not use the name of the class and the name of

the attributes to do the binding. Here, the prefix attribute is called `pre`, but it could have been called anything else. CDI doesn't use the name but the type. CDI knows that one attribute is of type `String`, another `long`, and the other `int`. That's why CDI is said to be strongly typed.

Listing 2.

```
public class NumberProducer {

    @Produces
    private String pre = "7";

    @Produces
    private int random =
        Math.abs(new Random().nextInt());

    @Produces
    private long post =
        Math.abs(new Random().nextLong());
}
```

Going back to our `BookService` class (**Listing 1**), the important information here for CDI is the datatype. Being strongly typed, CDI will inject the right value to the right datatype. This eliminates lookup using string-based names or XML for wiring so that the compiler will detect any errors. But as you might have guessed, datatypes are not enough. What if we need to inject another attribute of type `long` with a completely different value—for example, milliseconds? That's when we use qualifiers on producers.

Injecting qualified produced datatypes. If we need different values to be injected for the same datatype, we use qualifiers. In **Listing 3**, we change the `BookService` to add a qualifier on `postfix` and ask CDI to inject a 13-digit postfix. With a different qualifier, `@CurrentTime`, we tell CDI to inject the current time in milliseconds.

Listing 3.

```
public class BookService {

    @Inject
    private String prefix;

    @Inject
    private int random;

    @Inject
    @ThirteenDigits
    private long postfix;

    @Inject
    @CurrentTime
    private long millis;

    public Book createBook(String title) {
        String number =
            prefix + "-" + random + "-" +
            postfix + "-" + millis;
        return new Book(title, number);
    }
}
```

For this to work, we need to go back to the producer class and add the right qualifiers on the producers (see **Listing 4**). We can read this code as follows: produce a `String` called `@Default`, produce an `int` called `@Default`, produce a `long` called `@ThirteenDigits`, and produce another `long` called `@CurrentTime`. Being produced, they are all managed, and therefore injectable.

Listing 4.

```
public class NumberProducer {

    @Produces
    private String pre = "7";

    @Produces
    private int random =
        Math.abs(new Random().nextInt());

    @Produces
    @ThirteenDigits
```

```
    private long post =
        Math.abs(new Random().nextLong());

    @Produces
    @CurrentTime
    private long millis =
        new Date().getTime();
}
```

Producer methods. We've just seen field producers. The idea behind field producers is that we produce a field that becomes a managed object. But fields are limited, because we can initialize them only with simple values. What if we need to produce a more complex object? That's when we can use *method producers* instead. Method producers use the same idea: take a method, do whatever you need to do, and produce the returned value. This returned value is then managed by CDI. Another difference between producer methods and fields is that methods can have an `InjectionPoint` API as a parameter. This API provides access to metadata about an injection point.

As a simple example, we can take the exact same code seen in **Listing 4** that uses producer fields, and turn each field into a method. The `prefix` method returns a `String`, with the value 7, and this returned value is produced by CDI (**Listing 5**). Likewise for the `random`, `postfix`, and `millis` methods. Producer fields and producer methods are treated the same way. But this example does not show the power of producer methods. Let's look at a more complex example.

Listing 5.

```
public class NumberProducer {

    @Produces
    public String pre() { return "7"; }

    @Produces
    public int random() {
        return Math.abs(new Random().nextInt());
    }

    @Produces @ThirteenDigits
    public long post() {
        return Math.abs(new Random().nextLong());
    }

    @Produces
    @CurrentTime
    public long millis() {
        return new Date().getTime();
    }
}
```

More-complex producer methods. In **Listing 6**, the `FileProducer` bean has a `produceFile` method whose goal is to create a new file and return it. The algorithm here is much richer: the method takes the current directory in the file system and creates the `store` subdirectory below it. The file to be created on disk is called `file.txt`, and it is created if it doesn't already exist. If no exception is thrown, the file is returned and produced by CDI. As we can see in this example, a producer method lets the application take full control of the bean instantiation process—it acts as a factory. The returned file can now be injected anywhere.

Listing 6.

```
public class FileProducer {

    @Produces
    public Path produceFile() throws IOException {
      Set<PosixFilePermission> perms =
        PosixFilePermissions.fromString("rwxr-x---");
      FileAttribute<Set<PosixFilePermission>> attr =
        PosixFilePermissions.asFileAttribute(perms);
      Path directory =
        FileSystems.getDefault().getPath("store");
      if (Files.notExists(directory))
          Files.createDirectory(directory, attr);
      Path file = directory.resolve("myfile.txt");
      if (Files.notExists(file))
          Files.createFile(file, attr);
```

```
        return file;
    }
}
```

The way to use this produced file is easy. Take the `FileService` bean defined in **Listing 7**. It has a single method that writes some UTF-8 text to a given file. Instead of going through the long process of creating the file, it just injects it. A producer has already created the injected file, and it just needs to be injected so it can be used. Our system could then produce several files, just by using qualifiers. For example, we could create temporary files by qualifying the injection point with `@Temp`, or we could inject files used to store large binary objects with `@Lob`. As you can see, in this example, producer methods are very valuable. But producers are themselves CDI beans in their own right. So producers can use injection and other CDI features.

Listing 7.

```
public class FileService {

    @Inject
    private Path file;

    public void write(String text) throws Exception {
        Files.write(file, text.getBytes("utf-8"));
    }
}
```

Producers are CDI beans. Let us take the `FileProducer` bean and slightly change the file creation algorithm (**Listing 8**). This time, the directory to be created is injected. Being a managed bean, `FileProducer` can inject values like any other bean. In this example, it injects the root directory using the `@Root` qualifier (which has been produced somewhere else). The filename also changes by adding the number of milliseconds to the name. This number of milliseconds is injected and represents the current time. The file is created, returned, and produced, and it can now be injected.

Listing 8.

```
public class FileProducer {

    @Inject
    @Root
    private Path directory;

    @Inject
    @CurrentTime
    private long millis;

    @Produces
    public Path produceFile() throws IOException {
      Set<PosixFilePermission> perms =
        PosixFilePermissions.fromString("rwxr-x---");
      FileAttribute<Set<PosixFilePermission>> attr =
        PosixFilePermissions.asFileAttribute(perms);
      if (Files.notExists(directory))
        Files.createDirectory(directory, attr);
      Path file =
        directory.resolve("myfile-" + millis + ".txt");
      if (Files.notExists(file))
        Files.createFile(file, attr);
      logger.info("File {} created", file);
      return file;
    }
}
```

InjectionPoint API

Until now, the produced attributes and returned values that we have seen did not need any information about where they were injected. But there are certain cases where produced objects need to know something about the injection point into which they are injected. This can be a way of configuring or changing the producers' behavior depending on the injection point. CDI has an `InjectionPoint` API that provides access to metadata about an injection point. Thus we can create a producer method that has an `InjectionPoint` as a parameter.

Let's take for example the creation of a logger with the external Log4j library. Here, the `BookService` class creates a logger and sets its name to `BookService.class`.

```
public class BookService {

    Logger log = getLogger(BookService.class);

    // ...
}
```

If instead we wanted to inject the logger, we would need to produce it. For that, we take a separate `LoggingProducer` class, add a producer method that returns the logger, and inject it. It's as simple as that. But what if we need to reuse this logger in other classes? All the loggers will be named `BookService`, and that's not what we want. If `BookService` needs a logger called `BookService`, that means that `FileService` needs its own logger, as well as `ItemService`. The only parameter that changes here is the name of the logger.

```
public class FileService {

    Logger log = getLogger(FileService.class);

    // ...
}
```

How would we produce a logger that needs to know the class name of the injection point? The answer is by using the `InjectionPoint` API. This API provides access to metadata about an injection point, and in our case, it is used to configure the logger. In **Listing 9**, the `LoggingProducer` class produces a parameterized logger, thanks to `injectionPoint.getMember().getDeclaringClass().getName()`. This will get the injection point class name (`BookService`, `FileService`, or `ItemService`) and produce a logger per class. The `InjectionPoint` API has several methods to return the bean type of the injection point, its qualifiers, or the object itself.

Listing 9.

```
public class LoggingProducer {

    @Produces
    public Logger produceLogger(
        InjectionPoint injectionPoint) {
            return LogManager
                .getLogger(injectionPoint.getMember()
                .getDeclaringClass().getName());
    }
}
```

Using this produced parameterized logger is straightforward: we just inject it and use it as usual. The logger's category class name will then be set automatically, thanks to the `InjectionPoint` API. We can now do the same thing for `FileService` and `ItemService`. The code is strictly the same, but each bean will be injected with its own parameterized logger. Writing that simple parameterized producer has saved us time. We don't need to retrieve the class name to set it on a logger in every class where we want to use it. For example,

```
public class BookService {

    @Inject
    private Logger logger;

    // ...
}
public class FileService {

    @Inject
    private Logger logger;

    // ...
}
```

Disposers

In the previous examples, I used producers to create datatypes or POJOs so they could be managed by CDI. Producers act as factories: they create manageable objects. We have created objects and didn't destroy or close them, because we didn't need to. But some producer methods can return objects that require explicit destruction, such as a JDBC connection, a Java

Message Service session, or an entity manager. For destruction, CDI uses *disposers*. A disposer method enables the application to perform the customized cleanup of an object returned by a producer method. Let's look at an example using JDBC.

Producing a JDBC connection. In **Listing 10**, the `JDBCConnectionProducer` class has a `createConnection` method that creates a JDBC connection. This method takes a Derby database JDBC driver, creates a connection with a specific URL, and returns an opened JDBC connection. This method is annotated with `@Produces`, meaning that the connection can now be injected.

Listing 10.

```
public class JDBCConnectionProducer {

    @Inject
    private Logger logger;

    @Produces
    private Connection createConnection()
        throws Exception {
        Class.forName(
          "org.apache.derby.jdbc.EmbeddedDriver")
           .newInstance();
        Connection conn =
           DriverManager.getConnection(
               "jdbc:derby:memory:myDB;create=true",
               "APP", "APP");
        logger.info("Connection created");
        return conn;
    }
}
```

Now let's take a `JDBCPingService` (see **Listing 11**) whose job is just to ping a database to see if it's up and running. The `ping` method executes a SQL statement on a JDBC connection—that is, the produced connection that we can now inject. This code is fine, but we need to close the JDBC connection. One solution is to do it in the `JDBCPingService` by calling the `close` method. But this doesn't look right: if one producer has created this connection, it should be its job to close it. That's the role of disposers.

Listing 11.

```
public class JDBCPingService {

    @Inject
    private Logger logger;

    @Inject
    private Connection conn;

    public void ping() throws SQLException {
        conn.createStatement().executeQuery(
            "SELECT 1 FROM SYSIBM.SYSDUMMY1");
        logger.info("Ping....");
        conn.close();
    }
}
```

Disposing a JDBC connection. So we go back to our `JDBCConnectionProducer` class and add a `closeConnection` method (see **Listing 12**). The role of this method is to terminate the JDBC connection. Notice that both methods here are `private` and can't be invoked by other classes; only CDI can manage them.

To be able to automatically invoke the `closeConnection` method, it has to be annotated with `@Disposes`. Destruction is performed by a matching disposer method. Each disposer method must have exactly one disposed parameter of the same type as the corresponding producer method returned type—here, `Connection`. The disposer method, `closeConnection`, is called automatically when the client context ends (`@ApplicationScoped`, `@SessionScoped`, `@RequestScoped`, and so on).

Listing 12.

```
public class JDBCConnectionProducer {

    @Inject
    private Logger logger;
```

```
    @Produces
    private Connection createConnection()
        throws Exception {
            Class.forName(
                "org.apache.derby.jdbc.EmbeddedDriver")
                .newInstance();
        Connection conn =
            DriverManager.getConnection(
                "jdbc:derby:memory:myDB");
        logger.info("Connection created");
        return conn;
    }

    private void
        closeConnection(@Disposes Connection conn)
            throws  SQLException {
                conn.close();
                logger.info("Connection closed");
    }
}
```

The way to use a produced and disposed connection is as expected. The `JDBCPingService` in **Listing 11** changes just by getting rid of the `conn.close()` statement. It injects the produced JDBC connection with `@Inject`, and the `ping` method uses it to ping the Derby user database. The `JDBCPingService` class doesn't need to deal with all the technical plumbing of creating and closing the JDBC connection or exception handling. Producers and disposers are a tidy way of creating and closing resources.

Qualifying producer and disposer. *Qualifiers* are a type-safe way to distinguish between several beans and injection points, and can also be used on producers. For example, let's say I have two databases: a user database and a purchase order database.

To produce and dispose connections for these two databases, we just qualify the producer and disposer (**Listing 13**) with the `@UserDatabase` or `@PurchaseOrderDatabase` qualifier.

Listing 13. (Full listing available in downloads.)

```
public class JDBCConnectionProducer {

    @Inject
    private Logger logger;

    @Produces
    @UserDatabase
    private Connection
       createUserConnection() throws Exception {
          Class.forName(
   . . .
    @Produces
    @PurchaseOrderDatabase
    private Connection createPOConnection()
        throws Exception {
            Class.forName(
 . . .
```

For a given producer, there can be only one disposer method that matches the bean type and the qualifier. Also, notice that a disposer method must be declared within the same class as the producer. In **Listing 14** (download only), I have enriched the `JDBCPingService` class so it can ping both user and purchase order databases.

Alternatives and Producers

The benefit of *alternatives* is to change the behavior of an application at deployment time, just by notation in the `beans.xml` file. Alternatives can be applied to beans, as well as producers, just by using the `@Alternative` annotation.

Let's examine an example. In **Listing 15**, the `PurchaseOrderService` injects a value-added tax (VAT) rate and a discount rate. Both attributes of type `Float` are used to create a purchase order. Given a subtotal, the `PurchaseOrderService` sets both rates and calculates the total of the purchase order. As you might have guessed by now, both the VAT rate and the discount rate are produced in a separate class. Thanks to the `@Vat` and `@Discount` qualifiers, both fields of the same datatype, `Float`, can be produced without any ambiguity.

Listing 15.

```
public class PurchaseOrderService {

    @Inject
    @Vat
    private Float vatRate;

    @Inject
    @Discount
    private Float discountRate;

    public PurchaseOrder
        createPurchaseOrder(Float subTotal) {
            PurchaseOrder order =
                new PurchaseOrder(subTotal);
        order.setVatRate(vatRate);
        order.setDiscountRate(discountRate);
        order.setTotal(subTotal + (
            subTotal * vatRate) - (
            subTotal * discountRate));
        return order;
    }
}
```

But now let's say I deploy the application in a different environment or country, and I need to change the values of both rates. The goal is to produce another VAT rate and make sure it is annotated with `@Alternative`. We do the same for the discount rate. We end up with a single class, `NumberProducer` (see **Listing 16**), producing a VAT rate (5.5 percent) and an alternative VAT rate (7.8 percent) within the same bean.

Listing 16.

```
public class NumberProducer {

    @Produces
    @Vat
    private Float vatRate = 0.055f;

    @Produces
    @Vat
    @Alternative
    private Float altVatRate = 0.078f;

    @Produces
    @Discount
    private Float discountRate = 0.0225f;

    @Produces
    @Discount
    @Alternative
    private Float discountRateAlt = 0.125f;
}
```

To enable these produced `@Alternative` values, we just need to add the `NumberProducer` to the `beans.xml` deployment descriptor (**Listing 17**). The `PurchaseOrderService` in **Listing 15** doesn't change. Depending on whether the alternative is activated or not, the VAT rate of 5.5 percent will be injected, or the alternative one of 7.8 percent.

Listing 17.

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    . . .
       version="1.1" bean-discovery-mode="all">
  <alternatives>
    <class>org.foo.bar.NumberProducer</class>
  </alternatives>
</beans>
```

Conclusion

CDI makes it very easy to convert third-party classes into injectable objects using the `beans.xml` file and a variety of annotations. The annotations are remarkably simple, and they support a wide range of programming needs. As I've shown here, CDI creates one of the easiest ways to build apps using loosely coupled third-party components. </article>

MICHAEL HEINRICHS

BIO

Part 2

Inside the CPU: the Unexpected Effects of Instruction Execution

How false sharing and branch misprediction can have unwanted effects on your code's performance

The first article in this series about the effects of modern chip design on programming focused on the memory system, specifically the cache hierarchy. In this article, Part 2, I extend our investigation to the issues of multithreaded access. In Part 3, I will look at the internal workings of the CPU itself.

False Sharing

In our first experiment, we will see a surprising effect that can occur when memory is accessed concurrently from different threads. The code in **Listing 1** contains an array of 17 integers and four methods that modify different elements of the array. The experiment consists of two test runs in which we run two of the methods concurrently on different threads. In the first test run, we will execute the methods `modifyFarA()` and `modifyFarB()`, which modify two array elements that are 16 elements apart. In the second test run, we will execute the methods `modifyNearA()` and `modifyNearB()`, which modify two adjacent array elements. The only difference is how far apart the modified array elements are. How does that affect performance?

Listing 1.

```
public final int[] array = new int[17];

@Benchmark
@Group("near")
public void modifyNearA() {
    array[0]++;
}

@Benchmark
@Group("near")
public void modifyNearB() {
    array[1]++;
}

@Benchmark
@Group("far")
public void modifyFarA() {
    array[0]++;
}

@Benchmark
@Group("far")
public void modifyFarB() {
    array[16]++;
}
```

On my laptop, a method call in the first test run takes about 3.6 ns, while in the second test run it takes 4.5 ns. In other words, if the array elements are farther apart, modifications are 25 percent faster. How can that be?

There are actually two puzzles to solve here. Why does it matter how far apart the elements are? And why do these

PHOTOGRAPH COURTESY OF DEVOXX

methods interfere with each other at all, even though they access different variables?

Watchful readers will notice the similarity to the initial example in the first article of this series. If elements of an integer array are 16 elements or more apart, they will be located in different cache lines. If they are closer to each other, chances are great that they will end up in the same cache line. Is the observed variation in behavior related to cache lines?

Indeed, it is. The memory system has no understanding of Java. It does not know about Java arrays and Java variables. It smallest unit is a cache line. If two threads modify the same cache line, they interfere with each other and it does not matter that we modified two different variables in the source code.

Why do two threads that modify the same cache line slow each other down? We have to fill out our model of the cache hierarchy to understand this effect. When two threads run in parallel long enough, they end up on different CPU cores. CPU cores do not share an L1 cache—each core has its own. (Note: How the L2 and L3 caches are shared between cores depends on the architecture. For example, a typical setup is that L2 cache is shared between adjacent cores while L3 cache is shared between all cores.)

As each core copies the cache line into its respective L1 cache and does the update to the variable, the core notifies the other cores of the update and tells them to freshen the L1 cache copy they own.

The MESI protocol ensures that the **system never works with two different versions of the same cache line.**

Modern computers use a variant of the Modified Exclusive Shared Invalid (MESI) protocol to synchronize the L1 caches. The protocols used today have been improved, but the basic principle remains the same. In the MESI protocol, every cache line that was loaded into the cache is in one of four states: *Modified*, *Exclusive*, *Shared*, or *Invalid*. The Modified state means that the cache has an exclusive copy of the cache line and has modified it. If a cache line is in the Exclusive state, it also means that it is an exclusive copy, but it has not been modified yet. A cache line is in the Shared state if there are copies in two or more caches, but none of them has been modified. An Invalid cache line is no longer valid and cannot be used.

Figure 1 shows the state changes during our experiment. **Figure 1a** shows the state changes during the first test run, when the array elements were farther apart and the threads accessed different cache lines. When one of the threads loaded a cache line into the cache, the line was marked as being Exclusive. When the thread modified it, the state switched to Modified. From then on, the cache line remained in the Modified state. Access to the cache line was fast, because the Modified state tells us that we can update the cache line

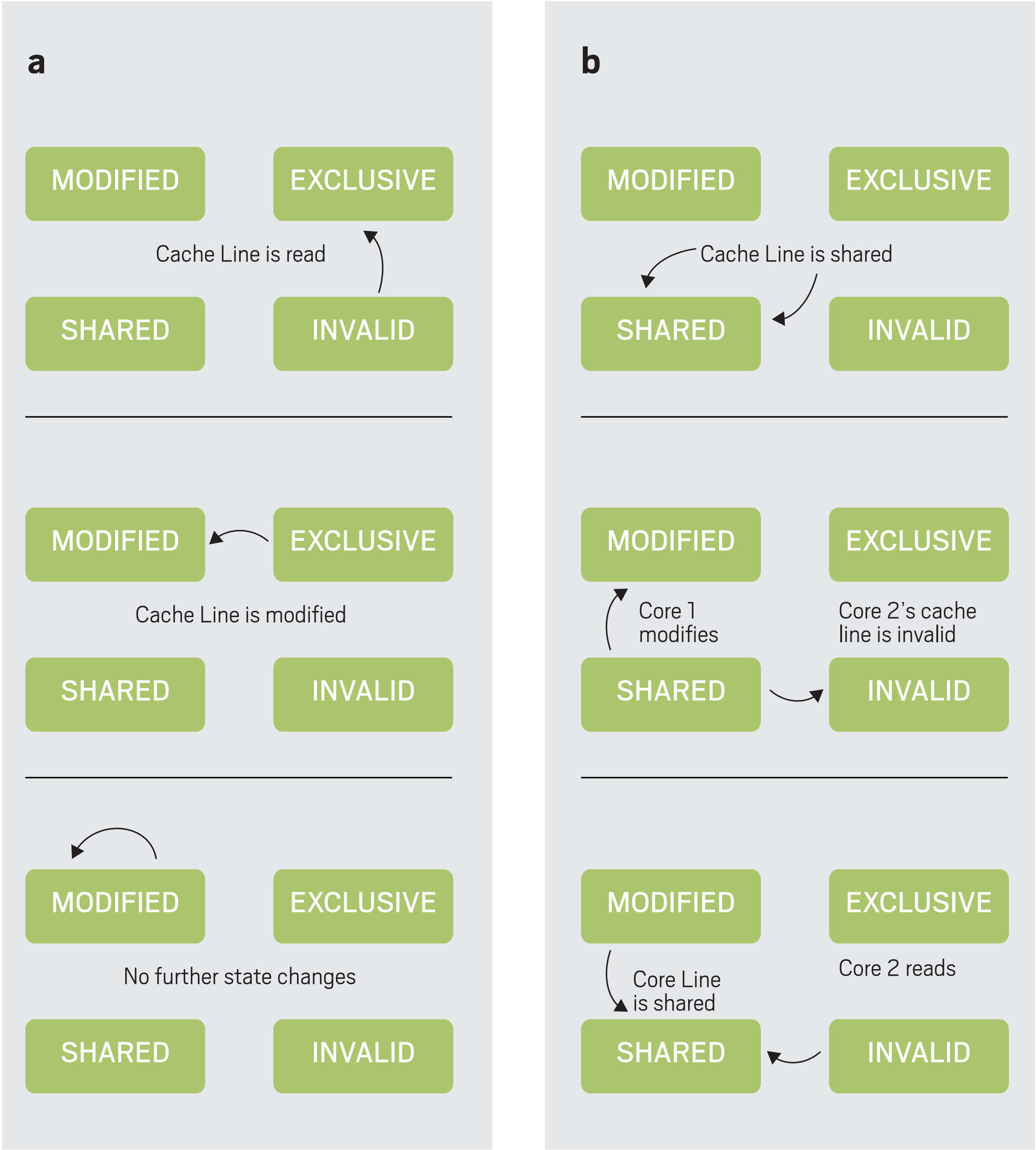


Figure 1: MESI state changes while modifying elements (1a: adjacent elements, 1b: distant elements)

directly without any further actions.

Things were more complicated during the second test run, as can be seen in **Figure 1b**. The MESI protocol ensures that the system never works with two different versions of the same cache line. Both threads modified the same cache line; therefore, when the second thread requested the cache line, it had to load it from the L1 cache of the first core and both caches had to mark their respective cache lines as shared. When one of the two threads tried to modify the cache line, it first had to signal the other cache that its cache line had become invalid. Only after that was successful and its cache line entered the Modified state was it able to do the modification. But when the other thread requested the cache line, both caches had to synchronize again and switch to the Shared state. Then we had to start all over again. The constant need to mark the other thread as invalid and synchronize back again once done was the reason why the second test ran slower than the first.

The problem of two seemingly independent variables affecting each other's performance is so common that it has its own name: *false sharing*. It is extremely tricky to detect, because false sharing is invisible if you look at Java alone. One of the examples in the Java Microbenchmarking Harness gives a good overview of different techniques to avoid false sharing between two variables. The basic idea of all solutions is to insert other variables to ensure that the two variables potentially affected by false sharing are located in different cache lines.

Linear Search or Binary Search? The Cost of Branch Misses

At this point, we leave the memory system behind to take a closer look at the CPU itself. With our next experiment, we will try to answer a simple question: If executed on a small array, which is faster—linear search (**Listing 2**) or binary search (**Listing 3**)?

Listing 2.

```
public static boolean linearSearch(
        int needle, int[] haystack) {

    for (int current : haystack) {
        if (current == needle) {
            return true;
        } else if (current > needle) {
            return false;
        }
    }

    return false;
}
```

Listing 3.

```
public static boolean binarySearch(int needle,
                                   int[] haystack) {

    int low = 0;
    int high = haystack.length - 1;

    while (low <= high) {
        int mid = (low + high) >>> 1;
        int midVal = haystack[mid];

        if (midVal < needle)
            low = mid + 1;
        else if (midVal > needle)
            high = mid - 1;
        else
            return true;
    }
    return false;
}
```

For large arrays, the binary search outperforms linear search by a wide margin: the larger the array, the larger the performance gap. But for small arrays, such as one of 16 integers, the answer is not straightforward, because the outcome depends on the CPU architecture. On my laptop, a binary search takes 28.2 ns on average, whereas a linear search takes only 21.8 ns. Most of us would have expected that binary search would be faster, because it requires fewer iterations. How is it possible that linear search is that much faster?

To get an initial idea, let's run perf again on both algorithms. (For a quick introduction to the usage of perf, see the first article in this series.) The per-

formance numbers are roughly the same, but there is a significant difference in the number of *branch misses*. *[Screen shots with full perf results are available in the download area. —Ed.]* The linear search had 346.9 million branch misses, while the binary search had 526.4 million. What is a branch miss? To understand branch misses, we need to take a step back and take a look at a mechanism called *pipelining*.

Pipelining. When the CPU processes an instruction, it actually has several steps to perform. The instruction needs to be fetched from memory and decoded. That is, the CPU must figure out what kind of instruction it is dealing with. After that, the instruction needs to be executed and the result has to be written back to memory. **Figure 2** shows the internal structure of a CPU and how a single instruction A steps through the different stages on a mythical CPU with no pipeline.

The operations performed at each stage are quite different. The stages are implemented in different parts of the CPU. Thus, if a processor really worked as shown in **Figure 2**, it would be inefficient, because most of its parts would be idle most of the time, waiting for the next instruction to arrive. Early on, chip designers thought about ways to keep all components busy and came up with a solution called pipelining.

The principle can be seen in **Figure 3**. Instruction A is fetched. While instruction A is decoded, the CPU fetches the next instruction, B. As it executes instruction A, the CPU decodes instruction B, fetches the third instruction C, and so on. Instead of processing one instruction after another, the CPU processes a stream of instructions: the instruction pipeline.

In a typical program, the CPU's rate of **successful branch predictions is well above 90 percent.**

This works extremely well. A single instruction still needs four cycles to step through all four stages, but with pipelining the throughput increases from one instruction every four cycles to one instruction per cycle.

Branch prediction. There is one case, however, in which pipelining becomes tricky: conditional jumps. Imagine that instruction A is a conditional jump and that, depending on the outcome, it may execute instruction B or jump directly to a distant instruction X. Should the CPU fetch instruction B or X while we decode A? It does not know the outcome of the conditional jump yet, because that becomes clear only after instruction A has finished the execution stage.

Different processor architectures deal with this situation differently. One of the most common strategies is *branch prediction*: The CPU guesses

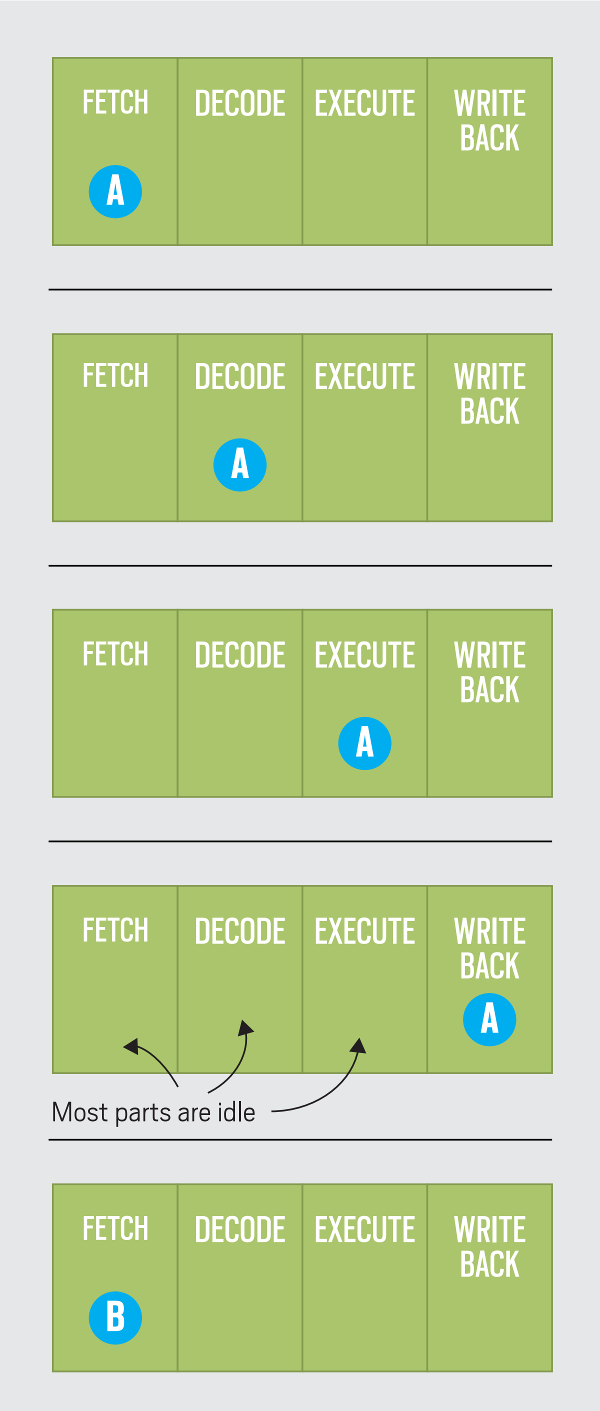


Figure 2: CPU execution without pipeline

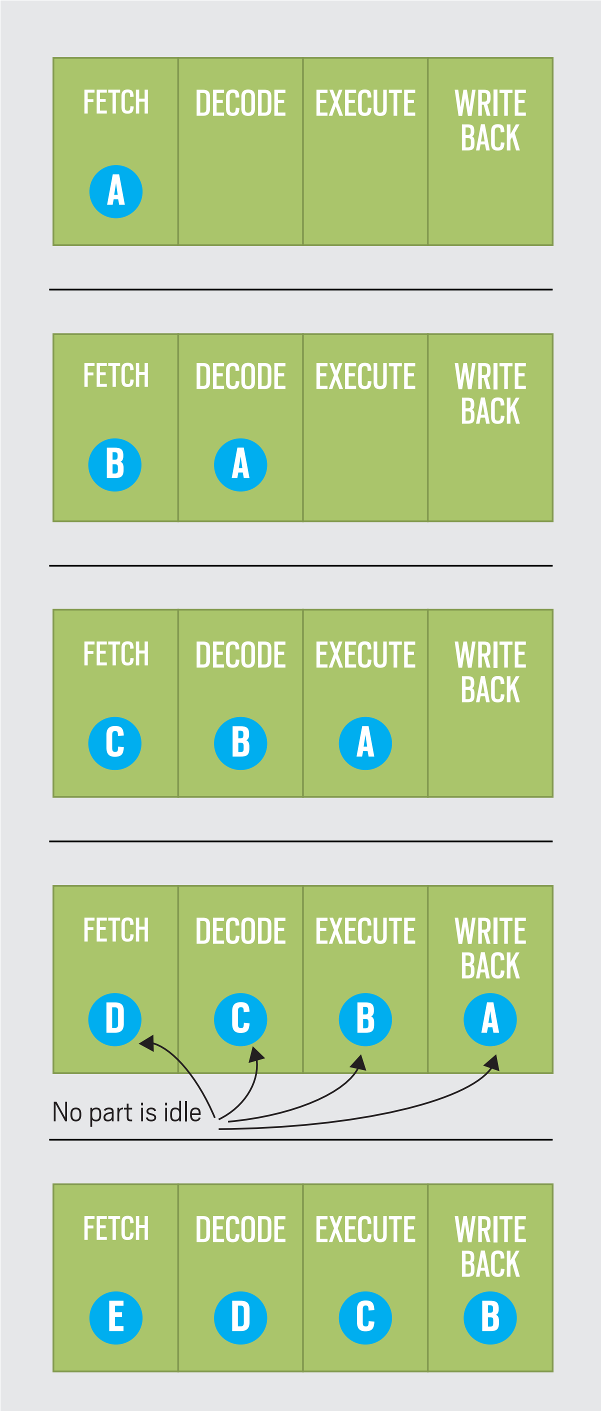


Figure 3: CPU execution with pipeline

which branch it should take. Even though the prediction algorithm has to be simple and fast, it is amazing how accurate it is. In a typical program, the rate of successful predictions is well above 90 percent.

What happens if the guess is wrong? The effect of a wrong guess and unnecessary execution is less dramatic than one might expect. Take a look at **Figure 4** and assume that instruction A is a conditional jump for which the CPU did a wrong guess. Once instruction A finishes the execution stage, the CPU knows that the branch prediction was wrong. It can simply drop instructions B and C now. At this point, they have only been fetched from memory and decoded. To the outside world, both operations have no noticeable effects. Therefore, the CPU can just throw out these instructions and continue fetching the first instructions of the correct branch.

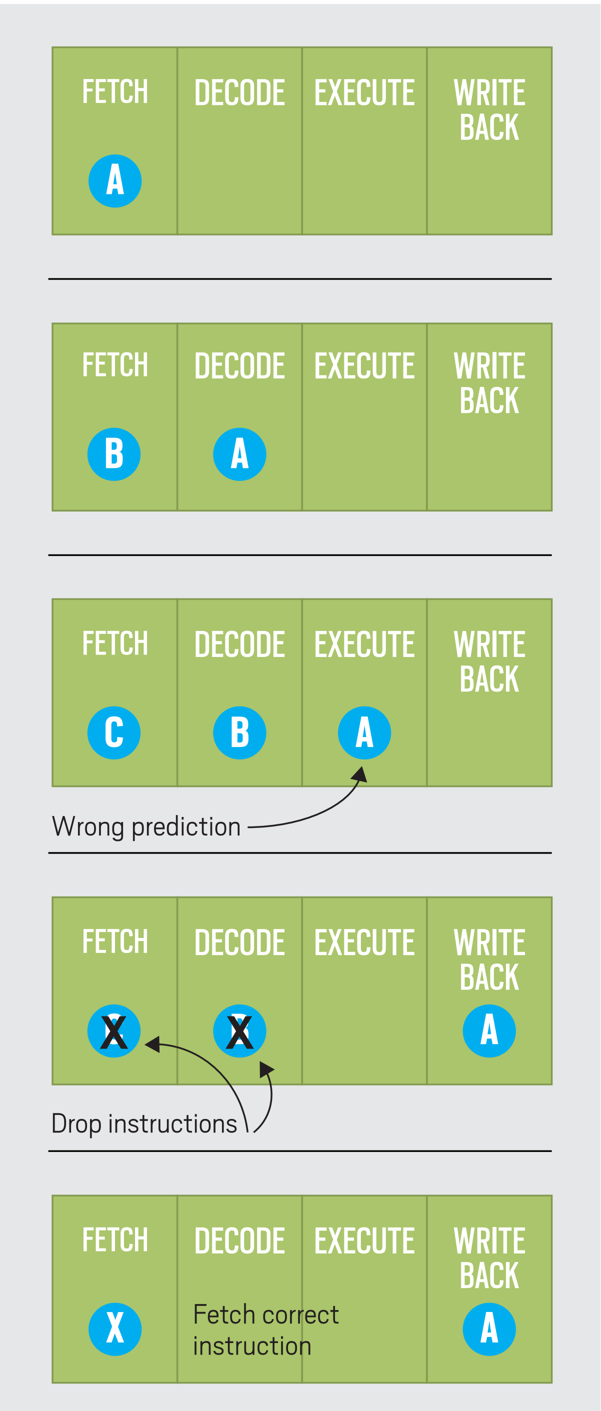


Figure 4: Branch misprediction

Branch prediction while searching. Nevertheless, branch mispredictions are expensive, because they nullify the performance increase we gained through pipelining our instructions. In the case of our two search algorithms, the performance loss is large enough to make binary search the slower alternative, even though it requires fewer comparisons to find the needle in the haystack.

The algorithm for linear search contains three conditional jumps: the `for` loop; the check for whether the element was found; and the check on the range of possible indexes. Each results in a conditional jump. Even though this is quite a few conditional jumps for such a short code segment, all of them are easy to predict correctly. The loop index is always smaller than the array bounds except at the very end, when the loop is exited. This means that once the loop is entered, every guess is likely to be right except the one at the very end. The same is true for the other two conditional jumps. If the CPU were to guess that the current element is not the element being searched for, it will always be right except once, when the element is found and the loop is exited.

The algorithm for binary search also contains three conditional jumps. Two of them are easy to guess right most of the time, for the same reason as described above. But in the loop, a decision must be made to continue in the upper half or lower half of the remaining array. The test data is evenly distributed, which means chances are 50-50 that the code needs to go to one half or the other. In other words, it is impossible to guess the right branch at this point. No matter which branch is guessed, it will be wrong half the time.

Conclusion

With the instruction pipeline and branch predictions, we have reached a level that you usually do not need to consider when optimizing your Java program. I have encountered maybe a handful of cases so far where avoiding branches actually did improve performance noticeably in very hot code segments. But pipelining and branch prediction are fascinating nevertheless. What this example shows is that for very small data structures, it often makes sense to use the simplest algorithm possible. The positive effects of clever but complex algorithms can easily be nullified by other effects. Therefore, it makes even more sense to adhere to the most important rule in software engineering: If in doubt, follow the KISS principle—that is, keep it simple. </article>

LEARN MORE

- Branch mispredicts using assembly on Intel processors
- "Eliminate False Sharing"
- Using padding in Java to avoid false sharing

FEATURED JAVA SPECIFICATION REQUEST

JSR 371: MVC 1.0 Specification

JSR 371: Model-View-Controller (MVC 1.0) Specification seeks to create a specification for an action-based MVC framework that works with a variety of view template languages, such as JavaServer Pages and Facelets. It also leverages the existing Java EE technologies you know and love, such as JAX 2.0; the Java API for RESTful Web Services (JSR 339); Bean Validation 1.1 (JSR 349); and Contexts and Dependency Injection (CDI) for Java 2.0 (JSR 365). The Java EE 8 Community Survey showed a strong community desire for additional MVC support alongside JavaServer Faces.

Specification Leads **Santiago Pericas-Geertsen** and **Manfred Riem** have joined forces with a collection of large and small organizations and individuals. If you'd like to comment on or contribute to JSR 371 or any other JSR, visit the JCP website and make your voice heard.

FEATURED JAVA USER GROUP: ANKARA, TURKEY

Ankara JUG was founded in October 2012 by **Barış Bal**, **Çağatay Çivici**, and **Mert Çalışkan**, who have worked together in the IT sector in Ankara, the capital of Turkey, for the last 10 years. Their goal was to stimulate Java user group (JUG) activity in Turkey.

The JUG holds meetups on the last Thursday of each month at Bilkent Cyberpark Dr. Fikret Yücel Konferans Salonu in Ankara. A typical meeting draws 100 people. The JUG has 1,000 registered users and is actively followed on Twitter and Facebook. The July event will be the 32nd consecutive monthly event. Çalışkan says, "We at the JUG are very proud of being so consistent. We have hosted many talented locals as well as international speakers, such as **Tyler Jewell** from Codenvy, **Reza Rahman** from Oracle, and **Serkan Özal** from Hazelcast."

In June the JUG hosted AnkaraJUG4Kids with the help of Oracle Academy. They used Oracle Alice to teach coding to children of ages 13 to 16. Çalışkan says, "It was fun to be there with the young generation, who will be a part of this digital world in the very near future."

The feeling at Ankara JUG is that Java is evolving fast, and they see opportunities for contributing to the ecosystem by adopting a JSR. As a result, they are also working to create groups and events from the community in Turkey for adopting a specification request.

For all the details, check out Ankara JUG's website, which is mostly in Turkish.

JAVA CHAMPION PROFILE

MARIO TORRE

***Mario Torre** has been an OpenJDK community member since its inception and is now an official member of the OpenJDK Adoption Group. In recent years, he has organized many of the famous Java DevRooms at FOSDEM.*

Java Magazine: Where did you grow up?
Torre: In Salerno, a beautiful city in the south of Italy between Amalfi and the Cilento Coast. I now live in Hamburg, Germany.
Java Magazine: When and how did you first become interested in computers?
Torre: When I was very young, my father bought me a Commodore 16. I was completely captured by it. I started to use PCs only when I was introduced to Linux, sometime in '97 or '98, although I owned a 386 for some time before that (which I used mostly to play X-Wing and Wing Commander).
Java Magazine: What was your first computer and programming language?
Torre: It was a BASIC dialect. I spent ages on the C16 getting the programs of the manual to work. I was introduced to C much later. One of my first real programs was a Norton Commander–like shell, which allowed users to organize files and programs. They could add shortcuts for up to nine of their favorite programs and launch them by simply pressing the associated number.
Java Magazine: What was your first professional programming job?
Torre: I always thought I would be a musician and programming was more of a hobby, so my first paid programming job came very late. It was Java-based and was something about organizing the lifecycle of an expedition, from arrival in the warehouse to delivery.
Java Magazine: What do you enjoy for fun and relaxation?
Torre: I usually play guitar and make music. But I also like model-making, drawing, reading, and creating my own music pedals. I'm working now on a digital delay on a Raspberry Pi.
Java Magazine: What "side effects" of your career do you enjoy the most?
Torre: Meeting people and traveling. I also deal a lot with students, introducing them to OpenJDK and the Java community. It's a lot of work, but I really enjoy it.
Java Magazine: Has being a Java Champion changed anything for you with respect to your daily life?
Torre: I think that being a Java Champion is a way to recognize one's work. In that respect, you just keep doing the same things as before. But of course there is the personal side of it: I need to remember that what I say will have an even greater resonance than before.
Java Magazine: What are you looking forward to in the coming years?
Torre: I want to see what will happen next and contribute to it. I believe we are doing a great job of keeping OpenJDK vibrant. I intend to keep on pushing Oracle toward openness. It's great to be part of this change.

Follow Mario Torre's blog or reach him on Twitter @neugens.

Comments

We welcome your comments, corrections, opinions on topics we've covered, and any other thoughts you feel important to share with us or our readers. Unless you specifically tell us that your correspondence is private, we reserve the right to publish it in our Letters to the Editor section.

Article Proposals

We welcome article proposals on all topics regarding Java and other JVM languages, as well as the JVM itself. We also are interested in proposals for articles on Java utilities (either open source or those bundled with the JDK). Finally, algorithms, unusual but useful programming techniques, and most other topics that hard-core Java programmers would enjoy are of great interest to us, too. Please contact us with your ideas at javamag_us@oracle.com and we'll give you our thoughts on the topic and send you our nifty writer guidelines, which will give you more information on preparing an article.

Customer Service

If you're having trouble with your subscription, please contact the folks at java@halldata.com (phone +1.847.763.9635), who will do whatever they can to help.

Where?

Comments and article proposals should be sent to me, **Andrew Binstock**, at javamag_us@oracle.com.

While it will have no influence on our decision whether to publish your article or letter, cookies and edible treats will be gratefully accepted by our staff at *Java Magazine*, Oracle Corporation, 500 Oracle Parkway, MS OPL 3A, Redwood Shores, CA 94065, USA.